2001

A temporal extension of formal concept analysis.

Rabih A. Neouchi
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A Temporal Extension of Formal Concept Analysis

by

Rabih A. Neouchi

A Thesis

Submitted to the Faculty of Graduate Studies and Research

through the School of Computer Science

in Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2001

Abstract

In artificial intelligence there is a great need to represent temporal knowledge and to reason about models that capture change over time. Change seems to be constant in a continuously changing world. In many domains such as science, medicine, finance, and demographics, change is noticeable from one time to another.

The thesis work aims at first extending FCA to capture temporal evolutions (TFCA) represented by concept lattices in time-stamped databases, and at applying the extended FCA techniques to data mining with an endeavor of inferring temporal properties.

Extending formal concept analysis to temporal domains allows us to use concept lattices to visualize temporal evolutions and deduce insights on the hidden regularities in the data.

To represent temporal evolutions, formal entities are time indexed. Temporal edges are added to concept lattices to show evolutions. Important temporal properties such as class evolution, persistence, and transition are classified and a mechanism for inferring them is presented. Algorithms for inferring temporal properties and generating temporal lattices from time-stamped databases are developed, implemented, and tested.

Dedication

I affectionately dedicate this thesis to my parents
Ahmad Tarek and Bassima Massri Neouchi

The "Grue" Property

The "grue" property is defined as:

> x is grue if and only if x is green and is observed before the year 2000, or x is blue and is observed after the year 2000.

This is a "weird" property but there is no obvious reason why we couldn't make up such a property. Now, let us pretend that the x referred to above are actually emeralds. Further, pretend that we have observed many emeralds and they have all been green and thus have had the property "grue". Then, intuitively, this should increase our belief that the next emerald we observe will be green and that it will be grue. This intuition is fine until New Years Eve in 1999. Now our pretended emeralds observed in 2000 should be grue and therefore blue and not green. Strange...Is it still strange if we pretend x are marbles rather than emeralds.

Goodman, Nelson. Fact, Fiction and Forecast. Indianapolis: Bobbs-Merrill, 1965.

Refer to Section 1.1, Page 2.

Acknowledgments

First, I want to thank God for all his bounties, favors and blessings upon me. This thesis would have never been completed without his mercy.

Second, I would like to thank all the people who, in one way or another, helped me complete my Master's degree. First, I am grateful to my advisors, Dr. Richard A. Frost and Dr. Ahmed Y. Tawfik, for their guidance and advice during the development of the research presented in this thesis as well as for their generous financial support. I am thankful to Dr. Frost for his kindness, and for introducing me to formal concept analysis and helping me turn this topic into an interesting research part of this thesis. I have been fortunate to have Dr. Tawfik as an advisor and to have the honor of working with him. I appreciate Dr. Tawfik's willingness to listen to my ideas, his patience and 'persistence' on conducting fruitful research, and his careful orientation and guidance during the late stages of my research that I spent mining information on temporal reasoning. I am deeply indebted to both of them for facilitating communication and for digging up dimensions in my mind that I would not have explored without my interaction with them. Undoubtedly, Drs. Frost and Tawfik are the best advisors any graduate student would ever want to work with.

I would also like to thank my other committee members. In particular, I am grateful to Dr. Walid S. Saba for offering me the chance to work on an exciting research problem, for his generous support and for his friendly encouragement. I admire Dr. Saba for his devotion to teaching and research. I am indebted to him for the warm learning environment he provided in the department. I also benefited a lot from the weekly Seminars on Artificial Intelligence (SAINT), and from the many technical and philosophical conversations it raised. I would also like to thank Dr. Alioune Ngom for his kindness, orientation, useful discussions and for the role he played as an internal reader. As well, I would like to thank Dr. Phil A. Graniero for agreeing to join my thesis committee and for his thoughtful insights about the thesis.

I want to express my gratitude to the office of graduate studies and research at the University of Windsor and to Drs. Frost and Tawfik, who provided me with a postgraduate tuition scholarship and a research assistantship funding respectively during the last year of finishing this thesis.

My life as a graduate student would have never been the same without the support of graduate colleagues at Windsor. I am thankful to Arslan Khan, Kamran Choudhery, and Nabil Abdullah for their friendship, advice and support throughout my graduate work, and for making the years I spent at Windsor very pleasant, instructive, and most rewarding. I am thankful to my friends Shawki Fattal, Rabih Halwani, Raed Sbeit, Ahmad El-Najjar, Abdul Rahman Ayoub, Maan Aziz, Dahir Ali, and Abdul Rahman Hussaini for being a constant source of encouragement and for their advice helping me avoid the obstacles of a graduate career. Since I stepped foot in Windsor and until the last minutes I spent there, Samir Massiss was more than a brother with the help and care he provided me with. Saving the best for last, I have been blessed to have my eternal childhood friend and high school classmate Bassel Baba near me in Detroit, Michigan. I very much enjoyed the weeks I spent at his place in summer 2000, and I will always value his concern and friendship for me no matter how far the distance separating us shall become.

Finally, I would like to thank my family for their patience and support throughout my studies. My parents, Ahmad Tarek and Bassima as well as my sister, Abir, and my brothers, Mustafa and Rami, always showed great interest in my work. Their constant encouragement, their pride in my success and their deep concerns for my difficulties are some of the major forces that helped me focus on my work and carry it through successfully. Mouhammad Farouk Neouchi, my uncle, was also very supportive and I much appreciate his advice on various aspects of life.

Table of Contents

List of Tables

List of Figures

Chapter 1
Problem Definition

1.1 Introduction

Human cognition has been the quest that most of the philosophers tried to tackle. Socrates, Plato, and Aristotle tried to represent conceptual structures of the human mind and to understand their hierarchies. Some of the epistemological questions they tried to answer were philosophical issues raised by the advent of cognitive scientific studies of the mind such as mental imagery, ultimate nature of mind, concepts, moral beliefs and knowledge. They studied causal laws that determine brain activity, addressed the requirement of having the same mind when a person retains his identity over time, and whether that person is the same person as he was as a child.

Contents of thoughts were also major concerns for the German philosopher and mathematician Gottlob Frege (1848-1925) [Cha94]. Frege [Fre92] studied the primary intension of a concept and stated that this intension fixes a reference in the actual world, and is generally cognitively accessible. He tried to solve puzzles that arise when the concepts involved have different primary intensions, and introduced senses to solve these problems. He required non-indexicality for sense reference, and proposed secondary intensions thus dividing the notion of sense into two distinct components. Frege recognized that intensions are central to the analysis of content, to deal with cognitive puzzles, and the fixation of reference.

Then, back in 1896, Charles Sanders Peirce developed his existential graphs as a way to treat conceptual structures with a graphical notation for first order logic and higher order extensions. Peirce's pragmatic philosophy aimed at reasoning in conceptual graphs and at using them as a graphical notation for logic, and at fusing both formal and practical meanings of concepts. His work reveals the importance of knowledge as a dynamic and

evolving entity, and aims at creating a process by which knowledge can be refined by a graphical expression. Peirce's pragmatic axiom can be seen in his expression: "Pragmatism is the principle that every theoretical judgment expressible in a sentence in the indicative mood is a confused form of thought whose only meaning, if it has any, lies in its tendency to enforce a corresponding practical maxim expressible as a conditional having as its apodosis in the imperative mood" [Pei31].

This monograph is about concept evolutions. It is about the kind of evolutions that are implied by the "grue" paradox [Goo65]. In the paradox, an object is "grue" if it is green before a certain time, T, and blue after T. An object is "bleen" if it is blue before T, and green after T. The paradox is how to tell the difference between something that is green and something that is grue. At the moment, or before T, it is impossible to know. After T, it will be easy. Similarly, it is easy to tell the difference between something that is green and something that is bleen before T. But after T, it might not be. Further, any system built on distinguishing between green and blue will work properly now with green and blue objects, but will fail after time T. We assume objects that are green today will be green tomorrow - in other words, we never observe grue objects. But the world might be full of grue objects, it's just that the time T hasn't come and gone yet. Grue may seem a bit contrived, but many things in the real world display similar discontinuities - a caterpillar discontinuously changing into a butterfly, or ordinary water changing to steam as its temperature is raised.

1.2 Thesis Statement

The thesis statement we are trying to defend in this monograph is that: 'Formal concept analysis can be readily extended to accommodate temporal knowledge', and that 'The temporal extension to FCA is useful for capturing temporal evolutions and for data mining applications'. The work described in this monograph includes:

- A summary of formal concept analysis.
- A discussion of temporal reasoning.

- An extension of formal concept analysis to handle temporal properties, which supports the thesis directly through construction.
- An analysis of this extension, which supports the thesis through arguments.
- Applications of use of the extension, which support the thesis through examples.

The last three points constitute our contribution to the work.

1.3 Research Objectives

As part of the research objectives in this monograph, we present two claims:

- Claim 1: 'formal concept analysis can be extended to a temporal formal concept analysis'.
- Claim 2: 'Temporal formal concept analysis is useful for capturing temporal evolutions'.

1.4 Research Hypotheses

For the first claim we present one hypothesis:

- Hypothesis 1: 'If we could encode temporal precedence into formal concept analysis, then we can extend formal concept analysis to represent temporal evolutions'.

For the second claim, we present the following two hypotheses:

- Hypothesis 2: 'Time-stamped databases can be used to extract temporal evolution patterns and thus play an important role in data mining'.
- Hypothesis 3: 'Objects evolve in time according to patterns'.

It is the purpose of this monograph to test and verify the validity of these hypotheses.

1.5 Methodology

This monograph is about a temporal extension of formal concept analysis. It is about capturing temporal evolutions in concept lattices and analyzing temporal metamorphosis, which we define as the change in the concept lattice over time. However, this work is more general than this and has more powerful features. In extending lattice theory over time, we try to discover trends from any kind of orders that need not be only 'time'. These orders could be for example, the different sequential levels of education or educational difficulty, the 'ph' level in a chemical reaction, the height from the ground for any given object, or a classification of time intervals/subintervals.

In this monograph, we present a method for handling the evolutions of concepts represented by concept lattices in time-stamped databases. Important temporal properties such as class evolution, persistence, and transition are classified and a mechanism for inferring them is proposed. The first step towards motivating our work is to show how the concepts that evolve with time induce a change in the concept lattice. To represent temporal evolutions, formal entities are time indexed, and temporal edges are added to concept lattices to show these evolutions.

1.6 Contribution

As mentioned earlier, the contribution of this work lies in proposing a temporal extension of FCA that supports the thesis statement through construction, an analysis of this extension that supports the thesis statement through arguments, and finally, applications of use of the proposed extension that support the thesis statement through examples. We present a new approach for representing temporal evolutions and inferring temporal properties from time-stamped databases. Most of the work that has been done in the area of temporal databases so far tried to answer temporal queries. This approach exploits some unique properties of patterns and trends that make the analysis and inferring process of these evolutions an efficient task. The idea was published in a paper [NTF01] that appeared in the proceedings of the 14th conference of the Canadian society for computational studies of intelligence that was held in Ottawa.

1.7 Thesis Structure

In the remainder of this chapter we present a general overview of this monograph. Chapter 2 reviews the fundamentals of formal concept analysis, introduces its mathematical foundation, and enumerates some of its current applications. The chapter also states some of its limitations, and hints at the need of the proposed temporal extension. At the end, related work to the proposed extension is described.

Chapter 3 introduces time, talks about the different representations of time proposed by James Allen, Drew McDermott, Patrick Hayes, and Andre Trudel, and links the bridge between formal concept analysis and temporal reasoning.

Chapter 4 motivates our work, describes the notation and addresses the representation used in our extension. The chapter also tackles temporal evolutions and their occurrences, states the advantages of understanding them, explores the new dimensions of applications of formal concept analysis that this extension creates, and describes temporal lattices and their types of edges. A classification of temporal properties and a mechanism for inferring them are proposed. Chapter 4 relies on our earlier work [NTF01] and emphasizes the role that temporal properties play in the process of defining the evolution patterns in the concept lattice.

New algorithms for inferring temporal properties, STEP, and for drawing temporal lattices, TLAT, are presented in Chapter 5. More specifically, the first algorithm tries to infer evolution patterns from the time-stamped database and passes this information to the second algorithm that generates the temporal lattices and draws the temporal edges as high as possible in the hierarchy of concepts to make them generic enough to describe evolutions of particular classes rather than specific objects. Time complexity of both algorithms is also studied and analyzed.

We proceed in Chapter 6, the final chapter, to draw conclusions about the research presented in this monograph, and talk about some possible applications of the proposed extension to formal concept analysis, and then proposing some possible extensions and alternative approaches which could be investigated in the future.

Chapter 2
Overview of Formal Concept Analysis (FCA)

2.1 Historical Background

Formal concept analysis (FCA) and conceptual graphs (CGs) have their foundations in the doctrine of the American logician and mathematician Charles Sanders Peirce (1839-1914). On the basis of his philosophical theory, we find a characterization of the formal context, the basic constituent of FCA [Sar99]. Peirce's main academic research was formal logic. He developed the logic of relatives that evolved into a first order logic by the middle of 1880[1]. The theory had a complete proof procedure in the form of existential graphs. Peirce also examined modal, temporal, and multi-valued logic. Later in the 1890's, Peirce studied set theory and formulated his famous theory of abduction. The theory of abductive reasoning is a logical way of inferring an explanatory hypothesis for a surprising phenomenon by relying on a guessing instinct to generate the explanation. Peirce's collected works [Pei31] were published after his death, and reflect the developments in his thinking.

Lattice theory plays a fundamental role in mathematics and, like group theory, is a fruitful source of abstract concepts to its different branches. It became an essential branch of modern algebra as a result of a series of articles published in 1933-7 by Garrett Birkhoff, Von Neumann, Ore, Stone, and Kantorovitch that showed that generalizations of Boolean algebra to suitable lattices had fundamental applications to modern algebra, projective geometry, point-set theory, functional analysis, logic and probability.

Birkhoff [Bir48] contributed to interesting discoveries in lattice theory, and tried to make its ideas accessible to mathematicians, to portray its structure, and to indicate some of its most interesting applications. The strength of lattice theory derives from the extreme

[1] The discovery of a first order predicate logic is attributed to Gottlob Frege (1848-1925). Whereas Frege might be accused of including disputable metaphysics in his theory, Peirce's theory was purely formal.

simplicity of its basic concepts and its general nature that pervade the whole of modern algebra. Lattices and groups provide two of the most basic tools of 'universal algebra'. The structure of algebraic systems is most clearly revealed through the analysis of appropriate lattices [Bir48].

Later on, the formal concept analysis (FCA) project started in the early 80's when a research group in Darmstadt, Germany, began to systematically develop a framework for lattice theory applications. It was first presented to the mathematical public in a programmatic lecture given by Rudolf Wille at the 1981 Banff conference on Ordered Sets [Wil82]. Since then, several hundred articles have been published on formal concept analysis and the Darmstadt group has participated in many collaborative projects.

2.2 FCA Features

FCA is a mathematical discipline whose features include:

- Visualizing inherent properties in data sets,
- interactively exploring attributes of objects and their corresponding contexts, and
- formally classifying systems based on relationships among objects and attributes through the concept of mathematical lattices. An example of a concept lattice is shown in Figure 2.1.

Concepts are necessary for expressing human knowledge. According to ([Wil82], [Stu98a] and [GW99]), the aim of FCA is a mathematical formalization of the concept 'concept'. Machine learning defines a concept as a set of objects with common structures. FCA aims at defining concepts and analyzing their hierarchy [WL97]. It automatically generates hierarchies called concept lattices that characterize the relationships among objects and their attributes.

FCA provides a formal, well-founded and easy-to-use framework in which we can automatically build a conceptual network that organizes all domain objects taking into account all its characteristics [FF98]. It relies on the pragmatic philosophy of Charles

Sanders Peirce who claims that humans can only analyze and argue within restricted contexts with a reliance on pre-knowledge and common sense [SWW98].

FCA is an algebraic formalism allowing implications between attributes to be determined and visualized. FCA views the world as containing objects possessing attributes. A single unit of thought is a concept [CEG97]. It is also a formal method to structure and visualize information in order to make it intelligible and interpretable [Erd98]. The author also claims that FCA might be useful in helping knowledge engineers in the process of building domain models.

In conclusion, FCA ([GW99] and [Wil82]) is a mathematical tool for analyzing data and formally representing conceptual knowledge. FCA helps creating conceptual structures from data. Such structures consist of units, which are formal abstractions of concepts of human thought allowing meaningful and comprehensible interpretation.

2.3 Mathematical Foundation

FCA is based on ordered set theory, and most of its notions have their roots in lattice theory. Birkhoff [Bir48] clearly presents this connection. In this section we review some mathematical terminology of FCA. According to ([GW99] and [Pri98]), FCA starts with the definition of a formal context and a formal concept.

2.3.1 Formal Context

A formal (or dyadic) context is the triple $K = < G, M, I >$ where:

- G and M are sets
- The elements of G are called the formal objects
- The elements of M are called the formal attributes
- I is a relation (also called dyadic relation) between G and M, with $I \subseteq G \times M$

The relationship is written as gIm or (g, m) ∈ I and is read as 'the formal object g ∈ G has the formal attribute m ∈ M'. A formal context can be represented by a cross table that has a row for each formal object g, a column for each formal attribute m and a cross in the row of g and the column of m if gIm. An example of a formal context and its corresponding concept lattice is shown in Table 2.1 and Figure 2.1 respectively.

Objects/Attributes	female	juvenile	adult	male
girl	X	X		
woman	X		X	
Boy		X		X
man			X	X

Table 2.1 Human context database [Pri01]

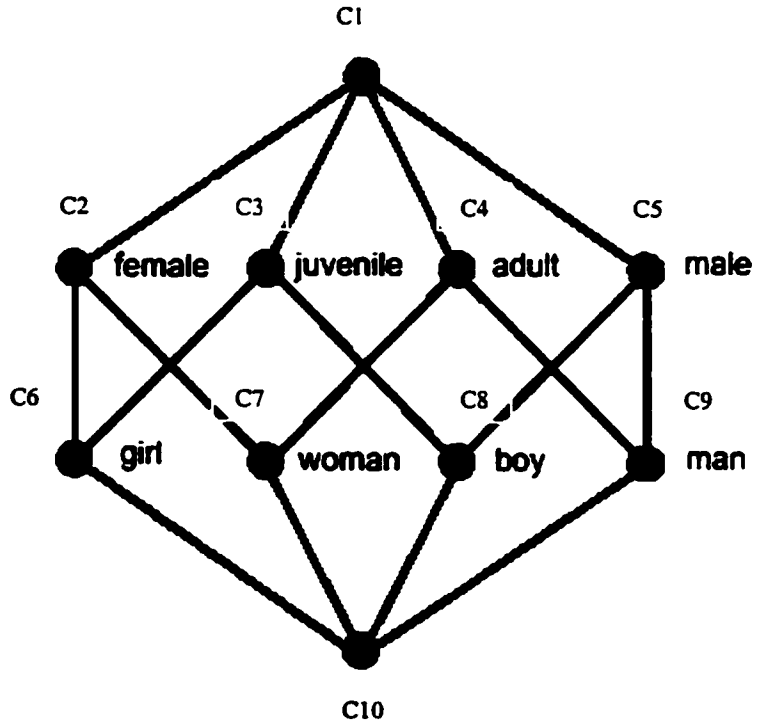

Figure 2.1 Human context concept lattice [Pri01]

2.3.2 Set of All Formal Attributes

Let K = < G, M, I > be a context. The set of all formal attributes of a set $A \subseteq G$ of formal objects is denoted by $A^{'}$ and defined by: $A^{'} = \{m \in M \mid gIm \text{ for all } g \in A\}$. Note that if $A = \{\varnothing\}$ then $A^{'} = A$. $A^{'}$ is also called the set of attributes common to the objects in A.

2.3.3 Set of All Formal Objects

Let K = < G, M, I > be a context. The set of all formal objects of a set $B \subseteq M$ of formal attributes is denoted by $B^{'}$ and defined by: $B^{'} = \{g \in G \mid gIm \text{ for all } m \in B\}$. Note that if $B = \{\varnothing\}$ then $B^{'} = B$. $B^{'}$ is also called the set of objects which have all attributes in B.

2.3.4 Formal Concept

A formal (or dyadic) concept c of the context K = < G, M, I > is the pair (A, B) with:

- $A \subseteq G$ and $B \subseteq M$
- $A^{'} = B$ and $B^{'} = A$

A is called the extent, e(c), and B is called the intent, i(c), of the formal concept c = (A, B). An example of a formal concept is shown in Table 2.2.

		B
A		XXXXX XXXXX XXXXX

Table 2.2 A Concept [GW99]

Note that if $K = < G, M, I >$ is a context, $A, A_1, A_2 \subseteq G$ are sets of objects, and $B, B_1, B_2 \subseteq M$ are sets of attributes, then:

- $A_1 \subseteq A_2 \Rightarrow A_2' \subseteq A_1'$
- $B_1 \subseteq B_2 \Rightarrow B_2' \subseteq B_1'$
- $A \subseteq A''$
- $B \subseteq B''$
- $A' = A'''$
- $B' = B'''$
- $A \subseteq B' \Leftrightarrow B \subseteq A' \Leftrightarrow A \times B \subseteq I$

The proofs can be found in be found in [GW99].

2.3.5 Smallest/Largest Formal Concept

For each formal object g the smallest formal concept to whose extent g belongs is denoted by γg. Similarly, for each formal attribute m the largest formal concept to whose extent g belongs is denoted by μm. The concepts γg and μm are called the object concept of g and the attribute concept of m, respectively.

2.3.6 Formal Subconcept

Let $c_1 = (A_1, B_1)$ and $c_2 = (A_2, B_2)$ be two concepts of a formal context $k = < G, M, I >$. The formal concept c_1 is a formal subconcept of the formal concept c_2 if any of the following equivalent propositions hold:

- $e(c_1) \subseteq e(c_2)$ or $A_1 \subseteq A_2$
- $i(c_2) \subseteq i(c_1)$ or $B_2 \subseteq B_1$

It follows from this definition that a formal concept c_1 is formal subconcept of the formal concept c_2 if c_1 has fewer formal objects and more formal attributes than c_2, and we write: $c_1 \leq c_2$ or $(A_1, B_1) \leq (A_2, B_2)$

2.3.7 Formal Superconcept

Let $c_1 = (A_1, B_1)$ and $c_2 = (A_2, B_2)$ be two concepts of a formal context $k = < G, M, I >$. The formal concept c_1 is a formal superconcept of the formal concept c_2 if any of the following equivalent propositions hold:

- $e(c_2) \subseteq e(c_1)$ or $A_2 \subseteq A_1$
- $i(c_1) \subseteq i(c_2)$ or $B_1 \subseteq B_2$

It follows from this definition that a formal concept c_1 is formal superconcept of the formal concept c_2 if c_1 has more formal objects and fewer formal attributes than c_2, and we write: $c_1 \geq c_2$ or $(A_1, B_1) \geq (A_2, B_2)$

2.3.8 Concept Lattice

WordNet ([Mil90] and [Fel98]) defines three senses for the word 'lattice', the first of which is 'an arrangement of points or particles or objects in a regular periodic pattern in two or three dimensions'.

In FCA, the set of all formal concepts of the formal context $K = < G, M, I >$ is denoted by ß(G, M, I). The relation $\leq$ is a partial order relation called the hierarchical order (or simply order) of the formal concepts. It is also called formal conceptual ordering on ß(G, M, I). A concept lattice of the formal context $K = < G, M, I >$ is the set of all formal concepts of K ordered in this way and is denoted by $\underline{ß}$(G, M, I).

It is important to note that one of the main advantages of concept lattices is that they can have fewer edges than a directed graph representing the same binary relation that the concept lattice represents. For example, suppose we want to add to Table 2.1 one object called *'young girl'* that has one attribute *'less than 12'*. One way to represent this binary relation into a directed graph is to add one node for *'less than 12'* and another one for *'young girl'* and then have 3 edges: the first would be between *'young*

girl' and *'less than 12'*, the second between *'young girl'* and *'female'*, and the third between *'young girl'* and *'juvenile'*. However, in order to represent this concept in the concept lattice, will only need to make *'girl'* a superconcept of *'young girl'* and have only two edges: one between *'young girl'* and *'less than 12'* and another between *'young girl'* and *'girl'*. In this case the concept lattice of Figure 2.1 would look something like Figure 2.2, where *'young girl'* inherits all the attributes that *'girl'* has, *'female'* and *'juvenile'*, in addition to having its own attribute *'less than 12'*.

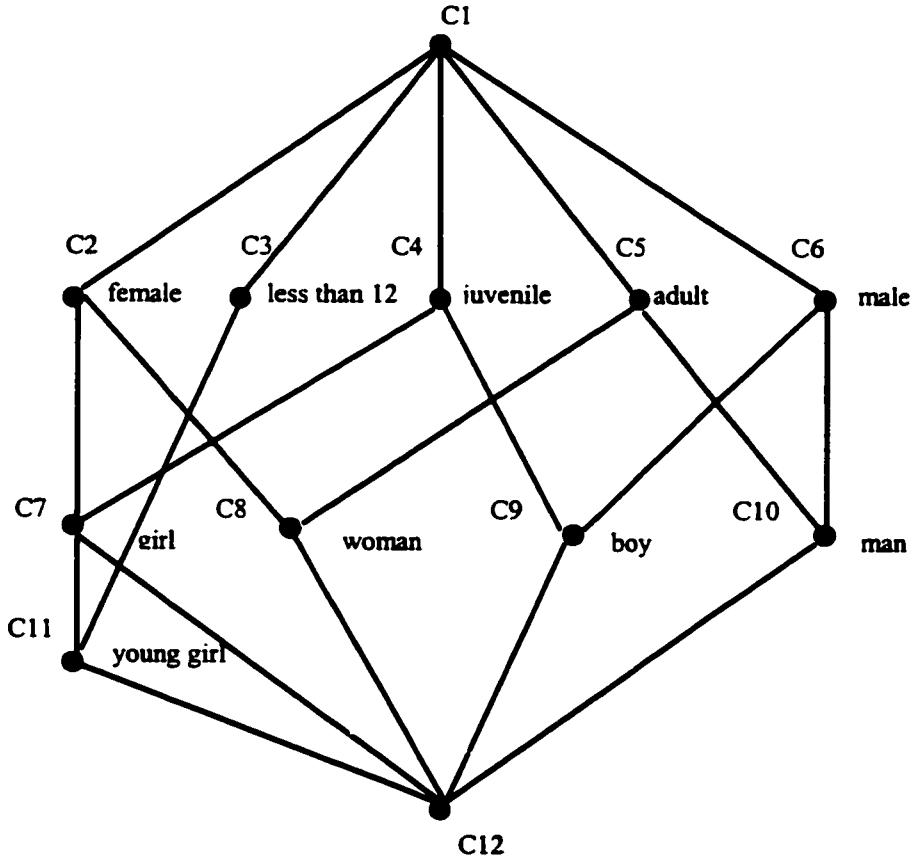

Figure 2.2 Modified human context concept lattice

A final note about the top-most and bottom-most concept nodes in the concept lattice, *C1* and *C12* in Figure 2.2, is that they correspond to the concept that consists of the union of all the attributes, and the concept that consist of the set of objects which have all the attributes respectively, with the possibility of the latter being an empty set, which is the case in Figures 2.1 and 2.2.

Another example of a concept lattice is shown in Figure 2.3, where the bottom-most concept node, *'Yosemite'*, is not an empty set. In fact, *'Yosemite'* is a subconcept of *'Lassen Volcanic'*, and therefore inherits all the attributes that it has. However, *'Yosemite'* has one additional attribute, *'Bicycle Trail'* that *'Lassen Volcanic'* does not have. In a similar fashion, if we want to know the park that has *'Swimming'* and *'Fishing'* attributes, a follow up of the corresponding links will lead to *'Channel Islands'*.

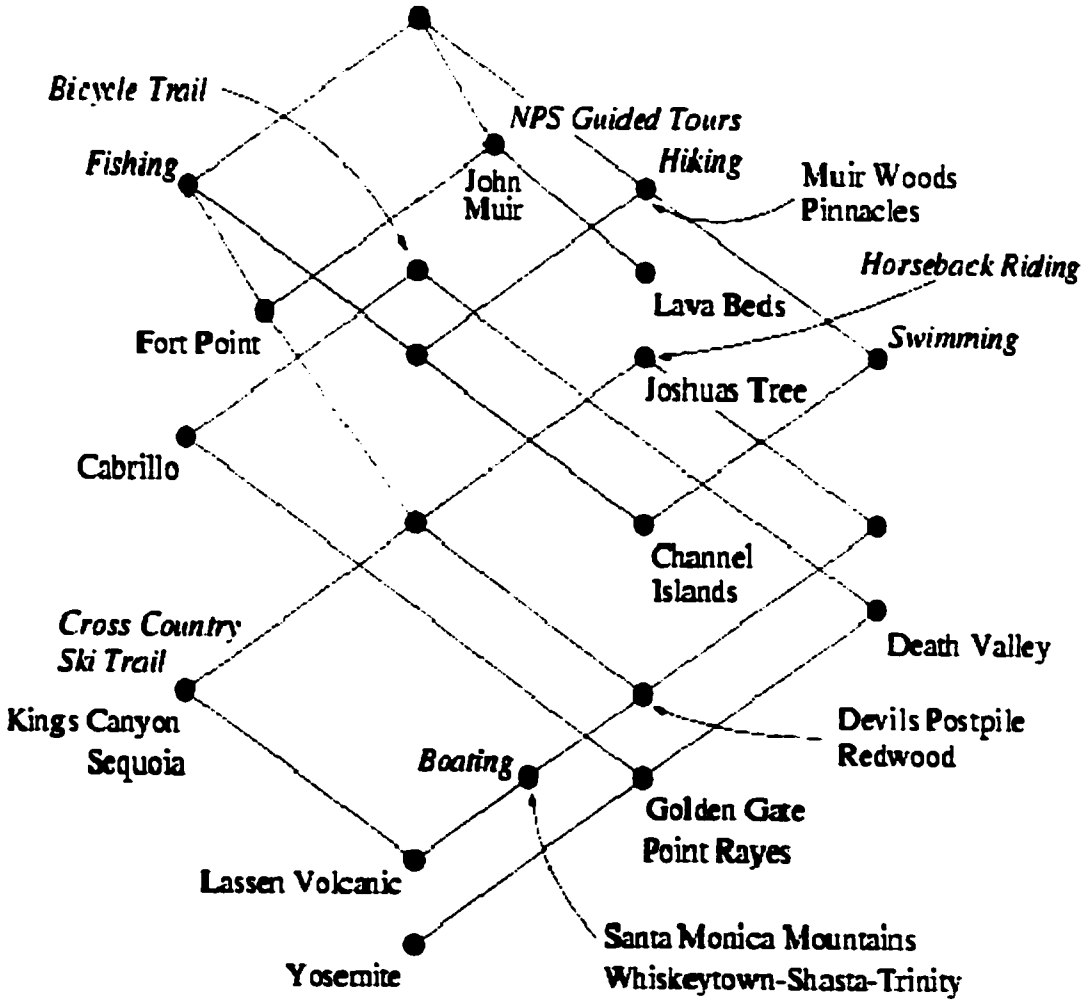

Figure 2.3 National parks concept lattice [Stu00]

2.3.9 Many-valued Context

A many-valued context $Q = < G, M, W, I >$ is a quadruple where:

- G, M and W are sets
- The elements of G are called the formal objects
- The elements of M are called the formal (many-valued) attributes
- The elements of W are called the attribute values
- I is a relation (also called ternary relation) between G, M and W (i.e., $I \subseteq G \times M \times W$) for which it holds that:

$$(g, m, w) \in I \text{ and } (g, m, v) \in I \text{ imply that } w = v.$$

The relationship is written as $(g, m, w) \in I$ or $m(g) = w$ and is read as 'the attribute m has the value w' for the object g. Q is called a n-valued context if W has n elements.

Like the one-valued contexts, tables, the rows of which are labeled by the objects and the columns labeled by the attributes, can represent many-valued contexts. The entry in row g and column m represents the attribute value m(g). If the attribute m does not have a value for the object g, there will be no entry. An example of a many-valued context can be found in Table 2.3. This context is used to show the different possibilities of arranging the engine and the drive chain of a motorcar (Figure 2.4).

Figure 2.4 Drive concepts for motorcars [GW99]

	De	Dl	R	S	E	C	M
Conventional	Poor	Good	Good	Understeering	Good	Medium	Excellent
Front-wheel	Good	Poor	Excellent	Understeering	Excellent	Very low	Good
Rear-wheel	Excellent	Excellent	Very poor	Oversteering	Poor	Low	Very poor
Mid-engine	Excellent	Excellent	Good	Neutral	Very poor	Low	Very poor
All-wheel	Excellent	Excellent	Good	Understeering/neutral	Good	High	Poor

De = drive efficiency empty; Dl = drive efficiency loaded; R = road holding/handling properties; S = self-steering effect; E = economy of space; C = cost of construction; M = maintainability

Table 2.3 A many-valued context: Drive concepts for motorcars [GW99]

2.3.10 Domain of an Attribute

Let $Q = < G, M, W, I >$ be a many-valued context. The domain of an attribute m is defined to be: $dom(m) = \{g \in G \mid (g, m, w) \in I \text{ for some } w \in W\}$. The attribute m is called complete if $dom(m) = G$. A many-valued context is complete if all its attributes are complete.

2.3.11 Conceptual Scaling

Concepts are assigned to a many-valued context by transforming it into a one-valued context in accordance with certain rules. The concepts of this derived one-valued context are then interpreted as the concepts of the many-valued context. This interpretation process is called conceptual scaling. Two steps are involved in the process of scaling. The first step is to interpret each attribute of the many-valued context by means of a context. This context is called conceptual scale. The choice of the scale for the attribute m is a matter of interpretation. The second step in the process of scaling is the joining together of the scales to make a one-valued context and to decide how the different many-valued attributes can be combined to describe concepts. In the simplest case, this is called plain scaling. An example of a derived one-valued context that corresponds to Table 2.3 is given in Table 2.4.

	De			Di			R			S				E				C				M			
	++	+	-	++	+	-	++	+	--	u	o	n	u/n	++	+	-	--	vl	l	m	h	++	+	-	--
Conventional			X		X			X		X					X					X		X	X		
Front-wheel		X				X	X	X		X				X	X			X	X				X		
Rear-wheel	X	X		X	X				X		X					X			X				X		
Mid-engine	X	X		X	X			X				X				X	X		X					X	X
All-wheel	X	X		X	X			X					X		X						X			X	

++ = excellent; + = good; - = poor; -- = very poor; u= understeering;
o = oversteering; n = neutral; vl = very low; l = low; m = medium; h = high

Table 2.4 A derived one-valued context [GW99]

2.3.12 Scale

A scale for the attribute m of a many-valued context is a one-valued context $S_m = (G_m, M_m, I_m)$ with $m(G) \subseteq G_m$ where:

- The objects, G_m, are called scale values
- The attributes, M_m, are called scale attributes

A more detailed interpretation of plain scaling, and the representation of the scales by concept lattices can be found in ([She99], [Pre97] and [GW99]).

2.3.13 Derived Context

A derived context (with respect to plain scaling) is the context $Q = < G, N, J >$ defined by $N = \cup_{m \in M} [\{m\} \times M_m]$ and $(g, (m, n)) \in J \Leftrightarrow (g, m, w) \in I$, and $(w, n) \in I_m$ [GW99]. The one valued context in Table 2.4 is obtained as the derived context of the many-valued context presented in Table 2.3 as the result of using the following scales:

$S_{De} := S_{Dl} :=$

	++	+	-
++	X	X	
+		X	
-			X

$S_R :=$

	++	+	--
++	X	X	
+		X	
--			X

$S_S :=$

	u	o	n	u/n
u	X			
o		X		
n			X	
u/n				X

$S_E := S_M :=$

	++	+	-	--
++	X	X		
+		X		
-			X	
--			X	X

$S_C :=$

	vl	l	m	h
vl	X	X		
l		X		
m			X	
h				X

Figure 2.5 Scales for Table 2.4 [GW99]

Had we used the scale S_E for the attributes De, Dl, and R, the derived context would have turned out slightly different. The concept lattice in Figure 2.6 corresponds to the 'Drive Context for Motorcars' shown in Table 2.4:

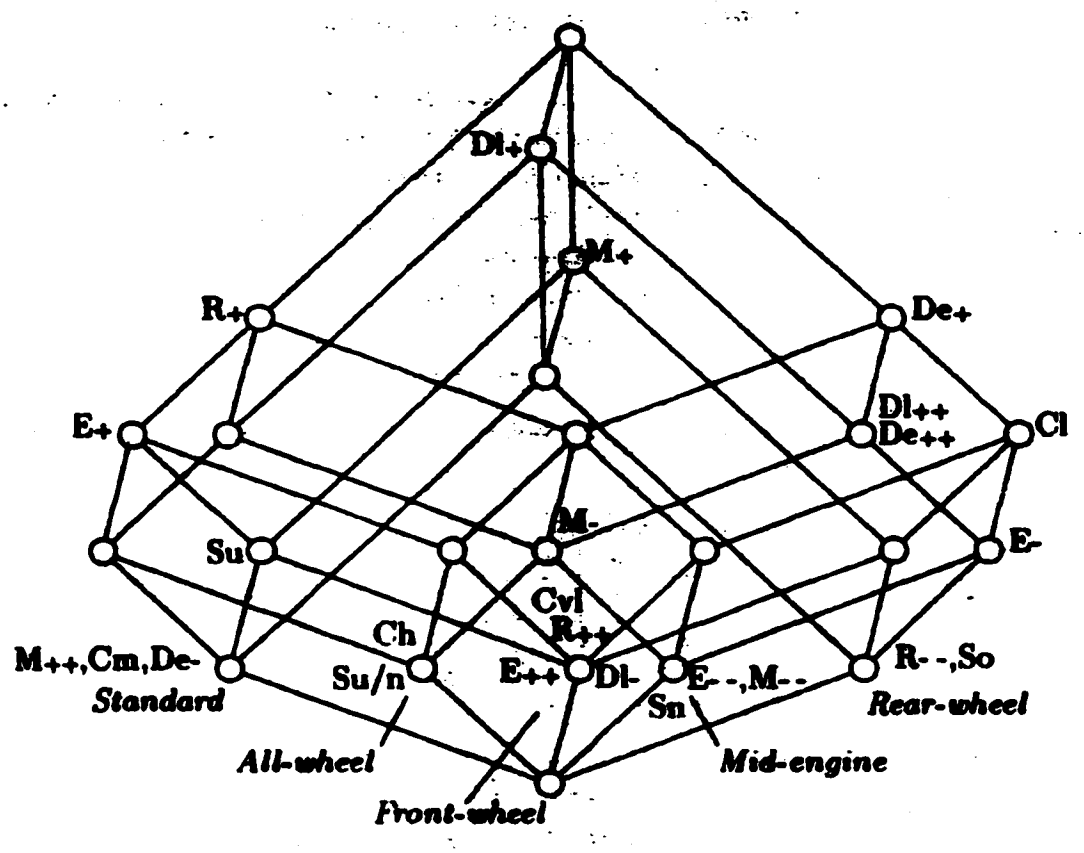

Figure 2.6 Concept lattice for the context of drive concepts [GW99]

2.3.14 Line/Hasse Diagram

A line diagram, also called Hasse diagram or concept lattice, is a graphical visualization of the concept lattice that allows the investigation and interpretation of relationships between formal concepts, objects and attributes [She99]. A line diagram contains the relationships between formal objects and attributes, and is therefore an equivalent representation of the formal context. It contains exactly the same information as the cross table. Dependencies and relationships between attributes can be easily detected in a line diagram. Algorithms for constructing concept lattices can be found in [She99].

2.3.15 Implications between Attributes

Implications between attributes are statements of the following kind: 'every object with the attributes *a*, *b*, *c* ... also has the attributes *x*, *y*, *z* ...'. Formally, an implication between attributes (in *M*) is a pair of subsets of the attribute set *M*. It is denoted by $A \rightarrow B$. When the sets are small or singletons, we omit the brackets and write $A \rightarrow m$ instead of $A \rightarrow \{m\}$. In dealing with relations between the attributes in the present context, one should examine the attribute implications that are true/hold only in this context. Concept lattices can be inferred/reconstructed from the implications between the attributes. Conversely, the implications between the attributes of a context can be read off the concept lattice. Since the systems of all implications between attributes that hold in a given context tend to be very large and to contain many trivial implications, it is sufficient to find subsystems that describe the concept lattice.

An example of an implication from the particular lattice shown in Figure 2.2 is *less than 12* $\rightarrow$ *{female, juvenile}*. However, if we had *'young boy'* in the lattice that has the attributes *'male'* and *'juvenile'*, then the implication would have been *less than 12* $\rightarrow$ *juvenile*. Another example of an implication from the lattice shown in Figure 2.3 is *Horseback Riding* $\rightarrow$ *{NPS Guided Tours, Hiking}*.

2.3.16 Triadic Concept Analysis

Triadic concept analysis is founded on a formal notion of triadic contexts, which allows set-theoretical formalizations. A triadic context is defined as the quadruple R = < G, M, B, Y > where G, M, and B are sets and Y is a ternary relation between G, M, and B (i.e. $Y \subseteq G \times M \times B$). The elements of G, M, and B are called objects, attributes, and conditions respectively and (g, m, b) $\in$ Y is read: 'the object g has the attribute m under (or according to) the condition b. The relational notation b(m, g) might also be used for (g, m, b) $\in$ Y. Just as two-dimensional cross tables describe dyadic contexts, three-dimensional cross tables, the rows of which are labeled by the

objects and the columns are labeled by the attributes and subtables conditions, may represent triadic contexts [LW95]. A triadic context gives rise to numerous dyadic contexts [Wil95].

2.3.17 Triadic Concept

A triadic concept of a triadic context $R = < G, M, B, Y >$ is defined as a triple (A_1, A_2, A_3) with $A_1 \times A_2 \times A_3 \subseteq Y$ which is maximal with respect to component-wise inclusion [LW95]. $A_i \subseteq K_i$ for for $i = 1, 2, 3$ and $A_i = (A_j \times A_k)^{(i)}$ for $\{i, j, k\} = \{1, 2, 3\}$ with $j < k$; A_1, A_2, and A_3 are called the extent, the intent, and the modus of the triadic concept (A_1, A_2, A_3), respectively [Wil95].

2.4 Applications

FCA has been applied to static domains such as assessing the modular structure of legacy code [Sne00] in software engineering, text analysis from different sources [EW98] in information retrieval, designing and exploring conceptual hierarchies of conceptual information systems [Stu99a] in knowledge acquisition, transforming object class hierarchies into normalized forms [SC99] in databases, performing structure-activity relationships [BB98b] in environmental/chemical applications, structuring the design interface of some educational applications [FF98], analyzing individual preference judgments [LW88] in decision making tasks, deriving a Natural Language Concept Analysis (NLCA) model [Sar99], improving the accuracy of speech recognizers [WNRR99], and extracting medical terms from discharge summaries that are used to structure medical thesauri [CE99B].

2.4.1 Text/Information Retrieval and Analysis

Applications of lattice theory offer potential usefulness in problems of operational bibliographic-information retrieval [Ped93]. The application of lattice theory, called relationship lattices, models both an extensible personal thesaurus and a convenient

user interface. This waives off the problems for non-expert users of bibliographic databases that are due to the complexities of query language and database structures, and to the differences between the user's terminology and the database's indexing terminology. The thesaurus could be subjective to declarative queries and extended into a knowledge base. A set of relationship lattice diagrams can represent the relationship lattice for a bibliographic information-retrieval interface. An advantage of this modeling is that it has an object-oriented program design. The aim is to support the user's browsing process, make querying and downloading of document records a convenient way, and avoid confusion by using windows and menus in the interface approach.

A new approach of applying FCA to the task of information retrieval is demonstrated [CE96]. Medical-discharge summaries are used as training data sets and the resulting concept lattices are structured. The set of objects is the set of sentences in the medical documents, while the set of attributes is the set of the concepts taken from a clinical and medical hierarchy. This clinical hierarchy is the result of building a corpus index first by incorporating SNOMED[2], Systematized Nomenclature of Medicine, into a formal concept lattice. The approach is novel since it emphasizes the use of both rich domain theories and expert knowledge to increase efficiency and performance, and proposes a mathematical solution to allow the user to submit and formulate a query and to navigate through the concept space displayed in the form of a concept lattice.

Analyzing texts from different sources, translating them to a formal knowledge representation, and merging them into a single coherent knowledge base/corpus are discussed in [EW98]. FCA is used in the analysis of this corpus to determine the reliability of information obtained from multiple sources, and then visually navigate this knowledge.

FCA has been used to explore information stored in a set of email documents [CE99a]. This is done with the use of a hierarchy of classifiers that extract key terms

[2] R. A. Côté, editor. Systematized Nomenclature of Medicine. Skokie Illinois, http://www.snomed.org

and regular expressions and associate attributes with emails. Hasse diagrams are used to represent the attributes. The conceptual scale is a subset of the attributes and can be used to construct a concept lattice showing the concepts generated by the emails and the attributes selected. The purpose is to encode implications and investigate them using a nested line diagram.

2.4.2 Knowledge Acquisition, Exploration and Discovery

The basic idea of concept exploration was already mentioned in the first paper on FCA [Wil82]. An overview over different exploration tools in FCA is given in [Stu96]. Specifically, Attribute exploration, Object Exploration, Concept Exploration, and Distributive Concept Exploration are discussed. Exploration tools in FCA are able to treat incomplete knowledge and have the criteria to complete acquired knowledge.

Formal representation of conceptual knowledge by means of FCA is discussed in ([Wil89] and [Wil92]). The central idea of knowledge acquisition in the frame of FCA lies in the assumption that conceptual knowledge can be represented by a formal context called a universe and its concept lattice. Knowledge exploration starts with a partial information about the universe and acquires more information by questioning experts. It is therefore important not to ask questions previously answered by the acquired knowledge. Two types of explorations are discussed: attribute exploration and concept exploration [Wil89]. Attribute exploration considers only all largest common subconcepts (infima) of the examined concepts and object exploration least common superconcepts (suprema) only. Concept exploration involves both attributes and object exploration and therefore treats largest common subconcepts and least common superconcepts equally [Stu98b]. A mathematical model for conceptual information systems is described in [Wil92].

To attain an optimization of conventional expert systems, FCA is used to develop an effective method of knowledge acquisition for a knowledge-based system [NK93].

FCA generates meta-rules that list dependencies between the rules' premises and make the execution time of the rule-based system practical.

TOSCANA is a graphical tool for exploring and analyzing data conceptually. It has been developed to assist the computation and graphical representation of conceptual structures [VW95]. Since only some attributes of a many-valued context are interesting for finding an answer to a specific question, the user can use TOSCANA to choose them and investigate the resulting nested line diagram with respect to his question. In certain retrieval system applications, TOSCANA has been extended to support components of knowledge inference and acquisition that need graphical tools to support the communication between the system and experts for the desired knowledge ([Wil89] and [Wil92]).

Knowledge acquisition tools of FCA are used for interactively exploring type hierarchies for conceptual graphs. Two interesting applications for Concept Exploration acquisition tool, one in FCA and another in conceptual graphs, are presented in [Stu97]. Distributive Concept Exploration is another knowledge acquisition tool in FCA similar to the more general Concept Exploration [Stu95]. Similarities and differences between Distributive Concept Exploration and Concept Exploration are given in [Stu98b]. Concept Exploration needs a heuristic for determining a suitable termination. Contrary to Concept Exploration, the algorithm for Distributive Concept Exploration will always terminate. This reduces the complexity of the exploration. The algorithm for Distributive Concept Exploration is described and illustrated in ([Stu95] and [Stu98b]), and implemented in C++ by B. Groh in [Gro95].

Conceptual information systems are based on the mathematical theory of FCA and their design involves both a domain expert and a knowledge engineer. Two principle ways for designing conceptual hierarchies of conceptual information systems, data driven design and theory driven design, together with their advantages and drawbacks, are discussed in [Stu99a]. Attribute Exploration, a knowledge acquisition

tool is applied to narrow the gap between both approaches and to design conceptual scales. Attribute Exploration determines implications between attributes in an interactive session. Since Attribute Exploration determines only the structure of the conceptual scale and does not indicate which concepts objects may label, a variation of Attribute Exploration called Clause Exploration can be applied [GK99].

FCA has been adopted in the Sisyphus-III experiment [SCTC96] that tries to explore the knowledge acquisition (KA) process [Erd98]. This leads to acquire a first impression of the concepts of the domain, which can be incorporated in a domain model. FCA helps in resolving potential problems such as the size of the data matrix, the lack of information in the resulting table and the contradictory statements.

A mathematical theory for knowledge acquisition, called attribute exploration, based on implications and counter examples is described in [Gan99]. The aim is to suggest a framework for including uncertain or background knowledge in the language of propositional logic. By using implications plus a background knowledge, the exploration itself will remain implicational. Despite the fact that the approach described has a high computational complexity and is limited to a specific kind of knowledge, it has a good potential for supporting mathematical knowledge processing and it is useful to support non-trivial research like finding axiom systems for relational structures used in linguistic classification.

2.4.3 Database Applications

An approach to transform object-oriented class hierarchies into a normalized form using FCA is presented in [SC99]. Some motivations of an object-oriented database schema are the support of a role concept, the support of multiple inheritance and the ability to model the universe of discourse (UoD). The framework of FCA is applied based on an intermediate representation for class hierarchies and is adapted to transform a schema into an object-oriented normal form. A concept lattice corresponds to a normalized description of a class hierarchy. The advantages of this

approach lie in the ability to consider both the extensional and intensional relationships thereby allowing a complete capture of the semantics of the schema, in building a concept lattice from a binary relationship existing between base extensions and attributes and in the support of the migration between differently structured databases.

2.4.4 Data Mining Applications

Data mining is the search for valuable information in large volumes of data [WI98]. It draws most of its principles from mature concepts in databases, machine learning and statistics. Weiss and Indurkhya [WI98] divide the types of data mining problems into two general categories: *prediction* and *knowledge discovery*, as shown in Table 2.5. Prediction considers specific goals that are related to past records with known answers, and uses them to foresee new cases. Knowledge discovery usually describes a stage prior to prediction in which knowledge is insufficient for prediction.

Prediction	**Knowledge Discovery**
Classification	Deviation Detection
Regression	Database Segmentation
Time Series	Clustering
Temporal Matching	Association Rules
	Summarization
	Visualization
	Text Mining

Table 2.5 Types of data mining problems [WI98]

The main goal of data mining is to analyze data sets and look for patterns and regularities that contain important knowledge about the data. In searching such regularities, a notion of similarity between objects and their attributes is needed. Searching for similar objects can help in prediction. Predictive data mining is the

search for very strong patterns in large databases and generalizing them to assist accurate future decision-making [WI98].

One possibility of using the similarity notions described in the previous two sections is the clustering of the attributes and event sequences by similarity. This helps in forming hierarchies of attributes that describe the structure of the data and inducing different kind of rules. Clustering of objects is important to discover the structure and relationships within data in many applications areas such as market basket data, telecommunications network data, medicine, biology, geography and chemistry. The need of computing similarities and forming these hierarchies arises from the fact that the domain expertise needed to form the hierarchy is not always available and from the need to derive similarity hierarchies on the basis of actual data, not on the basis of apriori knowledge. Three hierarchical clustering methods for clustering attribute and event sequences by using different similarity measures are considered in [Ron98].

Since concept lattices are the knowledge representation of FCA, they support the mining of association rules [Stu99b]. The aim is to motivate knowledge discovery support environments that integrate Conceptual Information Systems, which are based on FCA, and mining tools for association rules. The benefit of combining FCA and association rules is mutual and can enrich both theories.

Concept hierarchy plays a fundamental role in data mining [Lu97]. Knowledge discovery at the primitive level has been studied extensively and most existing statistical tools for data analysis are based on the raw data. However, abstracting data to a higher conceptual level and expressing knowledge at a higher conceptual level have superior advantages over data mining at the primitive level. Concept hierarchies organize data or concepts in hierarchical forms that are used for expressing knowledge in concise and high-level terms, facilitating the mining knowledge at multiple levels of abstractions. Concept hierarchies also have a fundamental role in data warehousing techniques in which the OLAP operations can be performed by concept generalization and specialization. The author also discusses the role of

concept hierarchies in the attribute-oriented induction (AOI) and multiple level rule mining, and emphasizes their use as important background knowledge in data mining, which was first introduced by [HCC93].

Stumme et al [STBPL00] address the problem of computing concept lattices from a data-mining point of view. According to them, Knowledge Discovery in Databases (KDD) is a current research domain that is common to both the AI and database community, where FCA has been used as a formal framework for implication and association rules discovery ([GM94], [PBTL99] and [TBPSL00]). The interaction of FCA and KDD in general has been discussed in ([HSWW00] and [SWW98]). We believe that TFCA will have more impact on KDD especially when it comes to dealing with temporal databases.

2.4.5 Chemistry/Physics/Environmental Sciences

In this kind of applications FCA is both a useful tool to classify classes of objects and a mathematical method to allow the checking of the derivation results of the 'if-then' rules. It is also advantageous in such domains due to its visualization and characterization of both the extensional and intensional data structures ([BB98b] and [BN97]).

Exploration of 31 calcium-aluminosilicate glasses data by means of FCA has been studied in [BN97]. FCA proved useful on two levels: the derivation of a classification of the glasses and the identification of units that serve as classifiers.

Concept lattice theory is also used in [BB98b] as an alternative way to perform structure-activity relationships. FCA provides an algorithm to relate structural information to environmental properties. By exploring a small chemical data matrix, the chemicals are the elements of the object set while sophisticated dependencies between different properties are the elements of the attribute set. To determine the concepts and discover the hierarchy of the underlying data set, conceptual scaling is

used to transform the many valued contexts, often described by ternary relations, into single valued ones.

An evaluation of five CD-ROM environmental databases by methods of lattice theory is shown in [BVS97]. The use of FCA is helpful in overcoming the difficulty of having several disconnected parts in Hasse diagrams. In this case Hasse diagrams are successfully used to perform comparative evaluation and to allow a clearer interpretation of the underlying data-set structures. The complexity in these fields is tremendous and often leads to many aspects of information. FCA is used here to perform generalized rankings and to seek deterministic arguments regarding goal functions.

2.4.6 Software Engineering

According to [Sne00], FCA is used for three purposes: to assess the modular structure of legacy code and modularize old systems, to analyze and explore configuration spaces, and to infer a transformed class hierarchy that is semantically equivalent to the original one but reflects actual member accesses in the program.

For the first purpose, the objects are the procedures of a program, the attributes are the global variables, and the goal is to find modules in legacy code by analyzing the relation between procedures and global variables. In nested local modules or procedures coupling is acceptable. Inferences, however, prevent modularization since they are a violation to the information hiding principle and can be automatically detected and removed. The concept lattice can both generate a modularization and serve as a quality metric. Modularization results in a partitioning of the variables, which can be found by lattice decomposition such as horizontal decomposition.

For the second purpose, a simple technique is the use of the C preprocessor CPP [KS94]. The objects are a set of code pieces while the attributes are a set of preprocessor symbols. FCA shows all dependencies between configuration threads,

which result in a low coherence and a strong coupling. FCA thus serves as a quality assurance tool for a good design. Simple data reduction techniques are implemented to allow insights into the structure of possible configurations and to visualize their quality according to software engineering principles.

For the third purpose, objects are all the variables and pointers while attributes are all data members and members form the program. The resulting lattice is interpreted as a class hierarchy that contains more classes but smaller objects than the old hierarchy. Lattices have a dynamic display according to the program behavior, and for reengineering purposes, should be simplified using semantics-preserving simplifications that are based on FCA and discussed in [ST98].

A framework for detecting design problems of class hierarchies and restructuring them is based on concept analysis [ST98]. The approach consists of first constructing a table that reflects the usage of a class hierarchy then deriving a concept lattice from that table. It is shown that the concept lattice makes the relationship between variables and class members explicit and serves as an interactive tool for redesigning and maintaining class hierarchies. The main goal of restructuring class hierarchies is code reuse to maximize both the sharing of expressions between methods and the sharing of methods between objects.

The potentials of concept analysis as an attractive foundation for a new class of program understanding and analysis tools and algorithms are explored in [Sne98]. Modularization of legacy code based on FCA is fully detailed in [LS97]. Additional applications of FCA in software engineering can be found in [She99].

Finally, a methodology of classifying, structuring and retrieving object-oriented classes based on FCA is described in [She00]. In this methodology, a class is classified and retrieved using type information and variable access behavior of methods available in the class, and then a class library is viewed as a many-valued

context. Concept-based Class retrieval is based on the concept lattice constructed from the class library, and reduces retrieval complexity [She00].

2.4.7 Education

FCA can be used to structure the interface design of some educational applications. In [FF98], FCA is used in two projects: an intelligent help system for the Unix operating system, Aran, and a multimedia tutorial for the written and oral comprehension system, Galatea. In order to formalize the linguistic conceptualizations used by specialists regarding their expertise domain applications, these domains have to be analyzed in terms of objects and their corresponding attributes. It has been shown that even in the complex applications where it is not easy to decide what the objects or their attributes are, FCA still gives a less subjective domain views than the other methodologies.

2.4.8 Decision Making

Concept lattices can be used to analyze individual preference judgments in the form of paired comparisons [LW88]. The method of paired comparisons is used to deduce preference judgments on a given set A of alternatives and to analyze dominance between objects. The formal context is usually called the tournament context and the concept lattice is obtained by structuring all the formal concepts by the subconcept-superconcept relationship. The line diagrams in the concept lattice are used to visualize the structure of the preference judgments and of the paired comparison data.

2.4.9 Statistics

In applications where formal contexts tend to be of a modest size, concept analysis experts learn from these contexts by visually inspecting the corresponding concept lattices. However, as formal contexts grow bigger, concept lattices become hard to analyze even with the support of tools like TOSCANA [VW95]. In these cases,

concept analysis is increasingly combined with statistical analysis where large contexts are constructed by a program ([Lin99], [SR97] and [ST98]). The resulting concept lattice is no longer inspected visually, but is part of an application's internal data structure ([Lin00] and [VML00]).

Various aspects of FCA from questions of classification up to theory of measurement have been discussed in the literature [FCA00]. FCA can be used as a conceptual clustering method ([Bis92], [Fis87a], [Fis87b], [Fis87c], [Gen89], [GLF89], [Leb86], [Leb87], [MS83a], [MS83b] and [SM86]), which is a method that generates descriptions of the clusters [Stu00]. In these methods, statistical techniques complement FCA techniques to capture patterns in the edit distance and similarity notion among clusters.

2.4.10 Natural Language

Conceptual structures in natural language (NL) are found as a result of the application of FCA to NL. Sarbo [Sar99] argues that, based on Peirce's semeiotic theory of linguistic knowledge about syntactic clusters, conceptual structures can be derived automatically in natural language leading to a novel model of language called Natural Language Concept Analysis (NLCA). For him a cluster is a characterization of a formal context, and is explained in the case of natural language in Peirce's semeiotic theory of signs [Pei31]. Based on this, three types of syntactic signs and corresponding relation schemes are developed to reflect conceptual distinctions that can be made in language, and to form the basis of NLCA.

2.4.11 Speech Recognition

FCA provides a way for improving tree-based state clustering which is the most well known approach for clustering context-dependent model-based speech recognizers [WNRR99]. The proposed method not only increases the accuracy of such speech recognizers in terms of limiting any additional phonetical knowledge but also

automates the construction of initial sets of triphones that limit the possible splits within the tree nodes and shows the usefulness of balancing the trees.

2.4.12 Medicine

Concept structures are differentiated mainly as factorial, hierarchical, and semantic attribute structures [Hul91]. Lattices of splitting and of quantity structures are discussed. Attribute structures in a clinical trial are also tackled.

Test data consisting of medical discharge summaries are used in ([CE96], [CEG97], [CEW98a], [CEW98b] and [CE99b]) to provide a corpus of evidential knowledge. Medical terms first extracted from the discharge summaries, then structured in a popular medical thesaurus, the National Library of Medicine's Medical Subject Heading (MeSH). This thesaurus contains a subsumption relation showing if a one medical term is a specialization of another.

2.5 Limitations

FCA has many advantages in knowledge processing including concept exploration, data mining, and rule discovery. When FCA deals with applications of a small number of objects and attributes, the complexity of the algorithms used for indexing and retrieving data is not a significant issue. However, when it is applied to explore large numbers of objects and attributes, the size of the data makes issues of complexity and scalability crucial [CE99b]. This is where conceptual scaling comes into the scene and shows its usefulness.

Concept lattices can grow exponentially in size with respect to their contexts ([Lin00], [NN97] and [STBPL00]). However, when their context tables are sparsely filled, they tend to grow quadratically at a slow rate with respect to their base relation. The density of context tables usually controls this growth rate. The main difficulty in dealing with galois lattice based systems comes from constructing the lattice itself [God89]. Therefore, it is

essential to keep the algorithmic complexity of the analysis procedures as low as possible, and for this purpose a parallel algorithm and another one that has an asymptotic time complexity that is linear in the number of concepts and the number of attributes have been proposed in ([NN97] and [VML00]) respectively.

Initially, FCA could not represent first order logic and capture quantification. From a theoretical aspect, the relations between FCA and first order logic have been widely studied, especially in [Zic91]. Bisson [Bis92] also used first order logic to represent conceptual clustering. However, the design and implementation of a complete first order FCA model was first proposed in [CM98] to improve the expression power of FCA as a knowledge-mining tool.

Lu [Lu97] mentions some problems related to concept hierarchies. These are the need for a basic technology that unifies the study of concept hierarchies, the possible types of concept hierarchies and their properties, the problem of automatically generating concept lattices since it is time consuming to construct a large concept hierarchy from a domain expert, and the need for a mechanism to realize efficient use of concept hierarchies in data mining.

Other FCA limitations include the lack of ways to capture precedence relationships between temporal points and intervals displayed by temporal lattices, and the incorrectness of abstractions that results in a wrong subconcpet/superconcept classification due to the fact that the latter depends on the number of attributes in the given context.

2.6 Related Work

A considerable amount of work has been done for extracting association rules from concept lattices ([Lu97], [PBTL98], [PBTL99] and [TBPSL00]). Discovering association rules, first introduced in [AIT93], is an important task in data mining and many of the algorithms that have been proposed in the literature. Common algorithms are the Apriori

[AS94] and Mannila's algorithms ([MTV94] and [Man97]) that use the subset lattice based approach for mining association rules. Another approach using the closed itemset lattice and described in [PBTL98] is closely related to Wille's concept lattice in formal concept analysis [Wil92]. This approach performs better in correlated data such as census data, and defines the semantics of association rules based on the galois connection operators. For example, it would be interesting to discover in a census database that '70 % of the persons who worked last year earned less than the average income', in a medical database that '80 % of people who have fever also have headaches', or in a market basket database that '90 % of customers who buy milk also buy sugar'.

2.6.1 FCA Representation of Association Rules

Formal concept analysis can be efficiently used for computing association rules in market basket applications. A market basket is a collection of items purchased by a customer in a single customer transaction [Ram98]. The input data of association rules algorithms can be written as a formal context *(G, M, I)* where *G* consists of the transaction IDs, *M* is the set of items, and the relation *I* is the list of transactions.

2.6.1.1 Important Measures

In the notion of market basket applications, we give the definitions of commonly used terms as well as two important measures: support and confidence ([AS95], [SA96], and [Ram98]):

- Transaction: is a set of items or an itemset
- Large itemset or litemset: is an itemset with a minimum support.
- Sequence: is a set of transactions.
- Support: the support for an itemset *i* is the fraction of customers who bought the items in *i* in a single transaction. The support for a set of items is the percentage of transactions that contain all of these items. The support for a rule $LHS \rightarrow RHS$, where both *LHS* and *RHS* are sets of items, is the support for the set of items $LHS \cup RHS$.

- Large Sequence: is a sequence satisfying the minimum support constraint. Each itemset in a large sequence must have a minimum support. Hence, any large sequence must be a list of litemsets.
- Data Sequences: is a set of sequences.
- Confidence: consider transactions that contain all items in *LHS*. The confidence for a rule $LHS \rightarrow RHS$, denoted by $conf(LHS \rightarrow RHS)$, is the percentage of such transactions that also contain all items in *RHS*. It indicates the degree of correlation in the database between purchases of these sets of items.

2.6.1.2 Association Rule

An association rule $X \rightarrow Y$ (with $X, Y \subseteq M$) is called *exact* if $conf(X \rightarrow Y) = 1$ and *approximate* otherwise. An exact association rule is also called an *implication*. For $X, Y \subseteq M$, an implication $X \rightarrow Y$ holds in the context iff the largest concept that is below all concepts generated by attributes in X is below all concepts generated by attributes in Y. In other words, the implication holds if each object having all attributes in X also has all attributes in Y; that is, an implication is an association rule with 100 % confidence [STBPL00]. In concept lattices, exact association rules can be directly read and visualized in the line diagram. It is also shown in [TBPSL00] how the association rules with less than 100 % confidence can be visualized in the line diagram.

Given the formal context and its corresponding concept lattice in Table 2.6 and Figure 2.7 respectively, then examples of implications in the concept lattice are shown in Table 2.7.

Nartional Parks in California	NPS Guided Tours	Hiking	Horseback Riding	Swimming	Boating	Fishing	Bicycle Trail	Cross Country Trail
Cabrillo Natl. Mon.						X	X	
Channel Islands Natl. Park		X		X		X		
Death Valley Natl. Mon.	X	X	X	X			X	
Devils Postpile Natl. Mon.	X	X	X	X		X		
Fort Point Natl. Historic Site	X					X		
Golden Gate Natl. Recreation Area	X	X	X	X		X	X	
John Muir Natl. Historic Site	X							
Joshua Tree Natl. Mon.	X	X	X					
Kings Kanyon Natl. Park	X	X	X			X		X
Lassen Volcanic Natl. Park	X	X	X	X	X	X		X
Lava Beds Natl. Mon.	X	X						
Muir Woods Natl. Mon.		X						
Pinnacles Natl. Mon.		X						
Point Reyes Natl. Seashore	X	X	X	X		X	X	
Redwood Natl. Park	X	X	X	X		X		
Santa Monica Mts. Natl. Recr. Area	X	X	X	X	X	X		
Sequoia Natl. Park	X	X	X			X		X
Whiskeytown-Shasta-Trinity Natl. Recr. Area	X	X	X	X	X	X		
Yosemite Natl. Park	X	X	X	X	X	X	X	X

Table 2.6 'National Parks in California' formal context [Stu00]

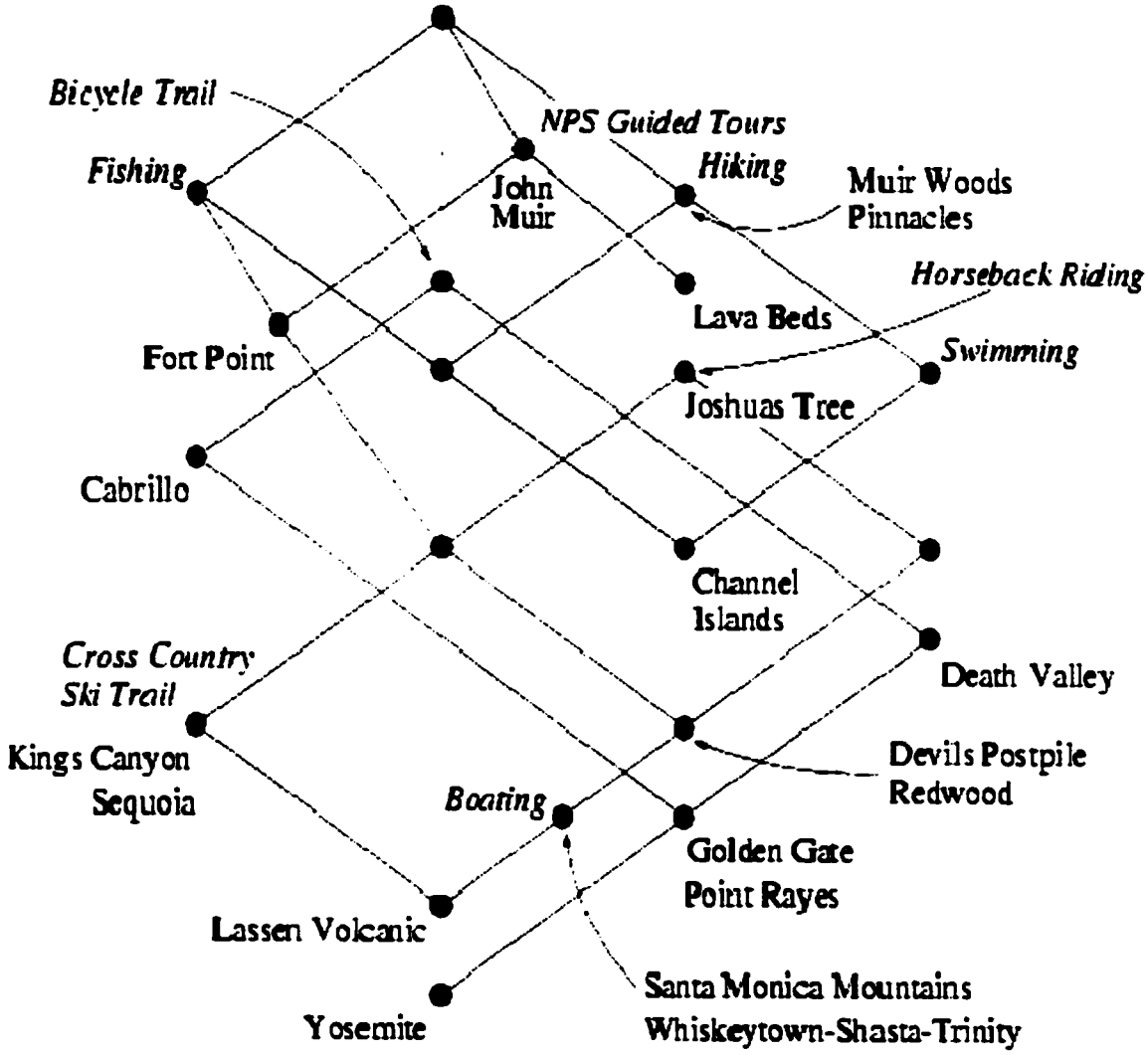

Figure 2.7 'National Parks in California' concept lattice [Stu00]

Implication	Support	Confidence
{Swimming} → {Hiking}	10/19 ≈ 52.6 %	100 %
{Boating} → {Swimming, Hiking, NPS Guided Tours, Fishing}	4/19 ≈ 21.0 %	100 %
{Bicycle Trail, NPS Guided Tours} → {Swimming, Hiking}	4/19 ≈ 21.0 %	100 %

Table 2.7 Implications in concept lattices [Stu00]

2.6.1.3 Support of an ItemSet

Composition: suppose $R1 \subseteq A \times B$, and $R2 \subseteq B \times C$, where A, B, and C are sets. Then the composition of $R1$ and $R2$, denoted by $R1 \,.\, R2$ is the relation $\{(x, z) \mid (x, y) \in R1 \text{ and } (y, z) \in R2\}$ [MKB86]. A *composed function* is a new function obtained through composition, which is a procedure of combining two functions.

Transitive Closure: suppose $R1 \subseteq A \times A$, where A is a set. The transitive closure of R, denoted by R^+ is $R \cup R^2 \cup R^3 \ldots \cup_{k \geq 1} R^k$. The *transitive reflexive* closure of R, denoted by R^*, is $R^+ \cup \{(a, a) \mid a \in A\}$ [MKB86].

If the composed function $'': \mathcal{B}(M) \rightarrow \mathcal{B}(M)$ is a closure operator on M, then the related closure system is the set of the intents of all concepts of the context or the set of all $B \subseteq M$ *with* $B'' = B$. This closure system determines the structure of the concept lattice. In the notation of FCA, the support of an itemset $X \subseteq M$ can be written as: $\sup(X) = \frac{|X'|}{|G|}$. In the case of X and Y with $X'' = Y''$, both sets have the same support. On the other hand, comparable attribute sets with the same support also have the same closures ([STBPL00] and [Stu00]):

- $X'' = Y'' \Rightarrow \sup(X) = \sup(Y)$.
- $(X \subseteq Y) \wedge (\sup(X) = \sup(Y)) \Rightarrow X'' = Y''$.

2.6.2 Pattern/Rule Discovery in Time Series

Time series applications arise frequently in financial and scientific domains such as the stock market, telecommunications data, medical signals, audio data, prediction data, and environmental sequence measures [DLMRS98]. Usually the interest is in finding rules relating the behavior of patterns over time within either one sequence such as 'a period of low telephone call activity is usually followed by a sharp rise in volume calls', or within two or more sequences such as 'if the Microsoft stock price

goes up and Intel falls down, then IBM goes up the next day'. The aim is not to define beforehand which patterns are to be used. Rather, the patterns are to be formed from the data in the context of rule discovery. In order to induce rules from the time series data, methods based on making sequences discrete and clustering them are used. In [DLMRS98], however, the rule-discovery method aims at finding local patterns from the series, in contrast to traditional time-series analysis that largely focuses on global models.

Similarity of objects is a crucial concept in several applications including biology, linguistics, logic, mathematics, philosophy, statistics, data retrieval, and data mining since it describes how far from each other two data objects are, and helps in finding patterns or irregularities in the data. Time series are an important class of complex data objects, for which similarity is nontrivial to define. There has been a lot of interest in querying time series on the basis of similarity [DGM97]. It is important to search within a time-series database for those series that are similar to a given query sequence. Two sequences are similar if they exhibit similar behavior for a large subset of their length. Similarity measures of sequences should be resistant to changes in the error measurement and in the scaling factors within the sequences. Algorithms for computing the similarity between sequences, as well as some generalizations and specializations of the similarity concept are discussed in [DGM97]. These algorithms rely on methods from computational geometry to compute the similarity measure, and to speed up the algorithms for finding similar time series.

2.6.3 Attribute and Event Similarity

When discussing similarity and databases, similarity between attributes is another class of similarity and is of equal importance to the similarity between objects stored in the database. Both similarities between objects and between attributes are one of the central concepts in data mining and knowledge discovery.

Similarity measures can be user defined, but an important problem is defining similarity on the basis of data. Ronkainen [Ron98] discusses two kinds of similarities: similarity between attributes and similarity between event sequences. Attribute similarity has two approaches. The internal or traditional attribute similarity measure approach considers only the values of the binary-valued attributes being scrutinized for similarity, and the new external attribute similarity measure approach that Ronkainen [Ron98] introduces, and that also takes into account the values of the other attributes in the relation. Ronkainen claims that the external measure approach reflects certain types of similarities and shows more accurate and useful results than the internal measure approach does.

Another important form of data considered in data mining is sequential data that occurs in many application domains, such as biostatistics, telecommunications, and user interface design. Such data can be viewed as a sequence of events where each event has an associated time of occurrence. Analyzing event sequences yields important knowledge about the behavior of a system. Event sequences are thoroughly discussed in ([MR97], [MTV97] and [Ron98]), and are based on the intuitive idea that similarity between event sequences should somehow reflect the amount of work needed to transform one event sequence to another. This notion is formalized as an edit distance between sequences that is efficiently computed using dynamic programming methods ([Gus97] and [MR97]).

Chapter 3
Time and Existing Methods for Temporal Reasoning

This chapter bridges the gap between FCA and its applications in static domains from one side, and temporal reasoning and the temporal extension of FCA (TFCA) on the other side by introducing time and some temporal logics available to reason about it.

3.1 What is Time?

Trying to define what time is would be a philosophical quest. Change would be the only means to express time. According to the Physics dictionary, time is a dimension that enables two otherwise identical events that occur at the same point in space to be distinguished. The interval between two such events forms the basis of time measurement. For specific scientific purposes, intervals of time are defined in terms of the frequency of a specified electromagnetic radiation. According to the International Committee on Weights and Measures, the basic time unit known as a 'second' is defined as 9,192,631,770 cycles of radiation associated with the transition between the two-hyperfine levels of the ground state of the sesium-133 atom[3].

Hayes [Hay96] mentions six different senses of the word 'time'. The first denotes its status as a physical dimension, also known as 'time dimension'. The second refers to a temporal continuum or space, also known as 'time plenum', in which all the events did and still happen. It can be regarded as a time line that need not be linear. The third concept is of 'time intervals' that are located in the time plenum. The fourth notion is that of 'time points'. The fifth sense is that of an amount of time, also known as 'durations'. Note here the assumption that every interval has its own duration. The last concept is a 'position' in a temporal coordinate system that is appropriate to answer temporal queries.

[3] Encyclopedia Britannica, http://www.britannica.com

In contrast to an interval, a position needs not have duration. Time positions can be modeled as either points or intervals.

WordNet ([Mil90] and [Fel98]) adds another four senses of 'time' to the previous six found in [Hay96]. The first concept is the 'instance' or single occasion of some event as in 'he called four times'. The second one is a 'suitable moment' as in 'it is time to go'. The third is the 'clock time' as in 'the time is 10 o'clock'. The last sense is a person's 'experience' on a particular occasion as in 'he had a time holding back the tears'.

In general, time is a global dimension that extends every area of artificial intelligence. Prediction, decision-making, communication, motion control, planning, behavioral strategies, scheduling, real-time object recognition, obstacle avoidance, vision processing, and machine learning all require reasoning about time. Temporal reasoning (TR) is a subfield of AI that acknowledges this central role of time.

3.2 Representations of Time

Representation of temporal information, and reasoning about such information requires a language that can capture the concept of change over time and can express the truth of statements at different times [SG88]. The goal of TR is a general theory of time, a time-efficient temporal reasoning framework, and a formalization of a temporal logic with robust syntax and semantics [Sho87].

3.2.1 Allen's Theory

Allen ([All83] and [All84]) proposes a temporal logic that includes descriptions of both static and dynamic aspects of the world. The temporal logic described is based on temporal intervals rather than time points. Properties capture the static aspects, while occurrences capture dynamic ones. A property holds over stretches of time, while an occurrence describes a change over stretches of time. The class of occurrences is divided into two subclasses, processes and events. Processes refer to

activities not involving an anticipated result, and one cannot count the number of times a process occurs. Events, on the other hand, describe an activity that involves a product or outcome, and one can usually count the number of times an event occurs.

Allen uses three predicates in his first-order predicate-calculus temporal logic. The predicate *HOLDS(p, t)* asserts that a property *p* holds and is true during a time interval *t*. The predicate *OCCUR(e, t)* takes an event *e* and a time interval *t*, and is true only if *e* happened during *t*. Finally, the predicate *OCCURING(p, t)* verifies whether a process *p* is occurring over an interval *t*.

Allen defines thirteen possible relationships that can hold between any two temporal intervals X and Y. Each of these is represented by a predicate in the logic. These relationships are *before*, *meet*, *start*, *overlap*, *during*, *finish*, *equal* and their inverses, as shown in Table 3.1.

Relation	Depiction	Abbreviation
X < b > Y *Y < bi > X*	XXX YYY	b = before bi = after
X < m > Y *Y < mi > X*	XXXYYY	m = meets mi = met-by
X < o > Y *Y < oi > X*	XXX YYY	o = overlaps oi = overlapped-by
X < d > Y *Y < di > X*	XXX YYYYYY	d = during di = contains
X < s > Y *Y < si > X*	XXX YYYYY	s = starts si = started-by
X < f > Y Y < fi > X	XXX YYYYY	f = finishes fi = finished-by
X < e > Y	XXX YYY	e = equals

Table 3.1 Allen's intervals

3.2.2 McDermott's Theory

McDermott's theory [McD82] presented a first-order temporal logic that serves as a framework for programs that model present change and future possibility. His logic captures two main ideas: the openness of the future, and the continuity of time. The first idea is modeled by having many possible futures or a branching future, while the second idea is modeled by having continuous instances between any two instants.

The universe is an infinite collection of states. A state is an instantaneous snapshot of the universe. Every state has its date, which is its occurrence time. States are arranged into a tree structure called a *chronicle tree*. A chronicle tree is a way events might go. It shows a complete possible history of the universe, and allows a world evolution to be traced, as shown if Figure 3.1.

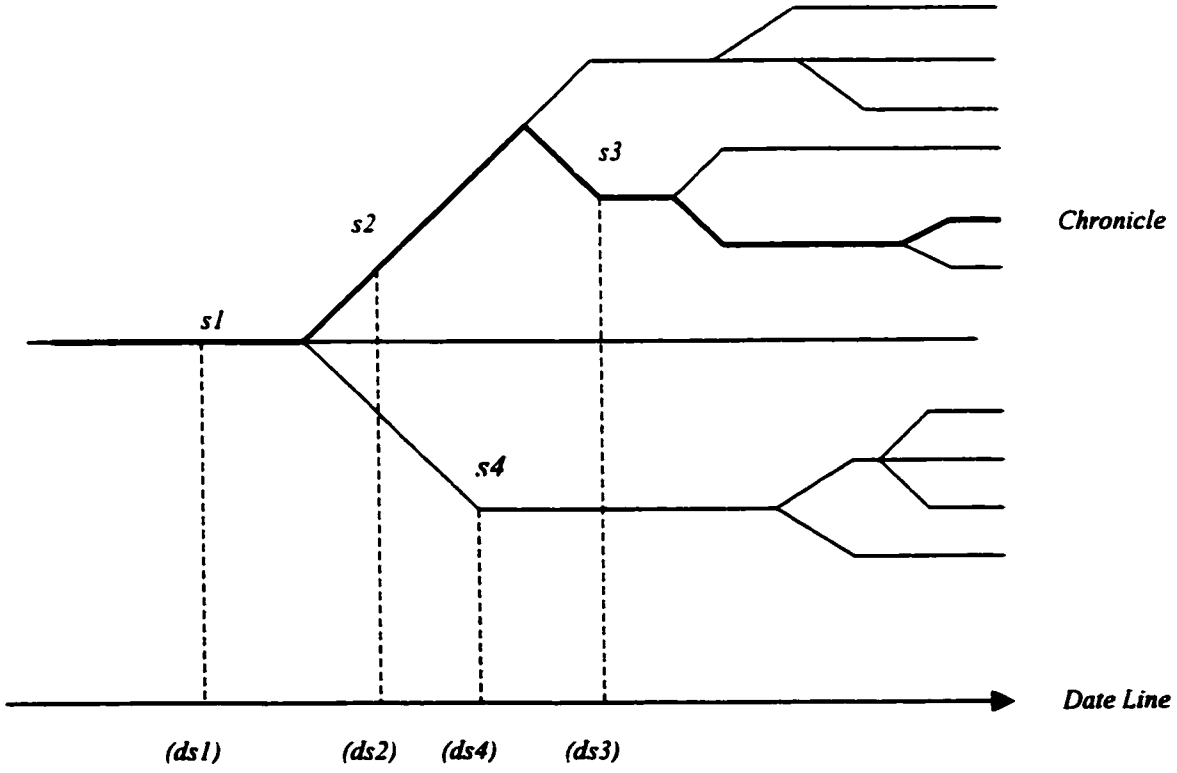

Figure 3.1 A tree of chronicles [McD82]

According to McDermott, states and chronicles are important because they introduce facts and events. Since a fact changes in truth-value over time, a fact is a set of states in which the truth-value of that fact is true. For example, *(ON A B)*, is the set of states in which *A* is on *B*. Note that *ON* in this context is not a predicate. As for events, McDermott emphasized the idea that events take time, and are not just fact changes as other AI researchers and philosophers claimed before. An event *e* is a set of intervals over which an event happens once, and occurs between states *s1* and *s2* by writing *(Occ s1 s2 e)*.

McDermott analyzes causality by stating that events can trigger either other events or facts after some delay *d* under a certain condition *c*. In the first case, the predicate *ecause* expresses that causation and has the following definition *(ecause c e1 e2 p d)* 'and read as '*e1* is always followed by *e2* after the delay *d* as long as the condition *c* is true'. The *p* parameter indicates whether the delay *d* is measured from the start of event *e1* or from its end. In the second case, the predicate *pcause* expresses the causation and has the definition *(pcause c e f p d l)* and read as 'event *e* is always followed by fact *f* after the delay *d* as long as the condition *c* is true'. Once *f* becomes true, it persists for lifetime *l*.

McDermott also examines the problem of reasoning about continuous change and planning actions. According to [Taw97], McDermott's theory does not offer an insight on the interval/subinterval relationship and cannot represent natural change.

3.2.3 Hayes' Theory

Hayes [Hay96] surveys several time structures and temporal theories, and develops a framework that integrates point and interval based temporal logics. For him, points can be thought of as intervals that are of zero length. He calls such a basic interval a *moment*, where moments cannot overlap or be contained in one another. He also divides temporal theories into three main categories.

The first is simple point axioms that describe a theory of time points ordering in which intervals are not mentioned, and whose structure can be described as either dense or discrete. Discrete linear points assume that there is an atomic spacing of time points that allows no closer divisions. The second is nonlinear time that results in the theory of branching point structure and allows every point to have infinitely many immediate successors and predecessors. Similarly, density or discreteness can extend this theory. The third is situation calculus time in which times or situations are partially ordered. Action axioms generate the structure of these situations. For example, *done* usually takes a situation and a sequence of actions and returns the situation resulting from *do*-ing those actions in that sequential order.

Allen and Hayes [AH89] present a temporal logic in which they represent Allen's thirteen interval relationships in terms of the '*meet*' relationship only. The definitions work by hypothesizing intervals that represent the gaps between the ends of the given intervals, and using auxiliary intervals to tie loose ends together. If '*i meets j*' is written as '*i:j*' and the dot notation represents the scope of the first-order predicate calculus operators, then Allen's thirteen relationships can be expressed as follows:

Relation	Depiction	Abbreviation	Definition
$X < b > Y$ $Y < bi > X$	XXX YYY XXXKYYY	b = before bi = after	∃ K . X:K:Y
$X < m > Y$ $Y < mi > X$	XXXYYY	m = meets mi = met-by	X:Y
$X < o > Y$ $Y < oi > X$	XXX YYY KLL LLM	o = overlaps oi = overlapped-by	∃ K, L, M . X = K:L & Y = L:M
$X < d > Y$ $Y < di > X$	XXX YYYYYY KKXXLL YYYYYY	d = during di = contains	∃ K, L . Y = K:X:L
$X < s > Y$ $Y < si > X$	XXX YYYYY XXXKK YYYYY	s = starts si = started-by	∃ K . Y = X:K
X < f > Y Y < fi > X	XXX YYYYY KKXXX YYYYY	f = finishes fi = finished-by	∃ K . Y = K:X

Table 3.2 The six relations between X and Y

An important aspect of Hayes' theory is that it tackles the issue of subinterval inheritance [Hay96]. In temporal theories, there are two views asserting the truth of a proposition in an interval. The first one entails that it is true at all points or subintervals of the interval, while the second explicitly denies the necessity of this subinterval inheritance and allows a proposition to be true during an interval without being true in all subintervals. Hayes reconciles these two views by proposing the *relativity of a proposition to an interval*, and distinguishing truth *on* an interval from truth *in* an interval. The truth of a proposition *on* an interval identifies that interval as an appropriate reference interval for the proposition, while truth *in* an interval means that there is a containing interval *on* which the proposition is true. Note that

subinterval inheritance of truth *in* an interval follows from the transitivity of the subinterval relation, and that truth *in* an interval need not entails truth *on* that interval.

3.2.4 Trudel's Theory

Trudel [Tru94] presents a two-dimensional temporal structure of a first-order logic that can capture intuitions about the past, present and future in contrast to most temporal first-order logics in artificial intelligence that have a linear temporal ontology. The main advantage of this two dimensional structure is the ability to record *when* knowledge is added or updated. It has an ever changing present, past and future relative to each present. As the present changes, the past and future change to reflect the continuous learning about the past and the revision of future plans.

The temporal ontology consists of a Cartesian plane on which the present moves along the line $y = x$. the line segment $\{y = x, x > p\}$ represents the actual future, $\{y = x, x < p\}$ represents the actual past, $\{y = p, x > p\}$ represents the expected future, and $\{y = p, x < p\}$ represents the perceived past, as shown in Figure 3.2.

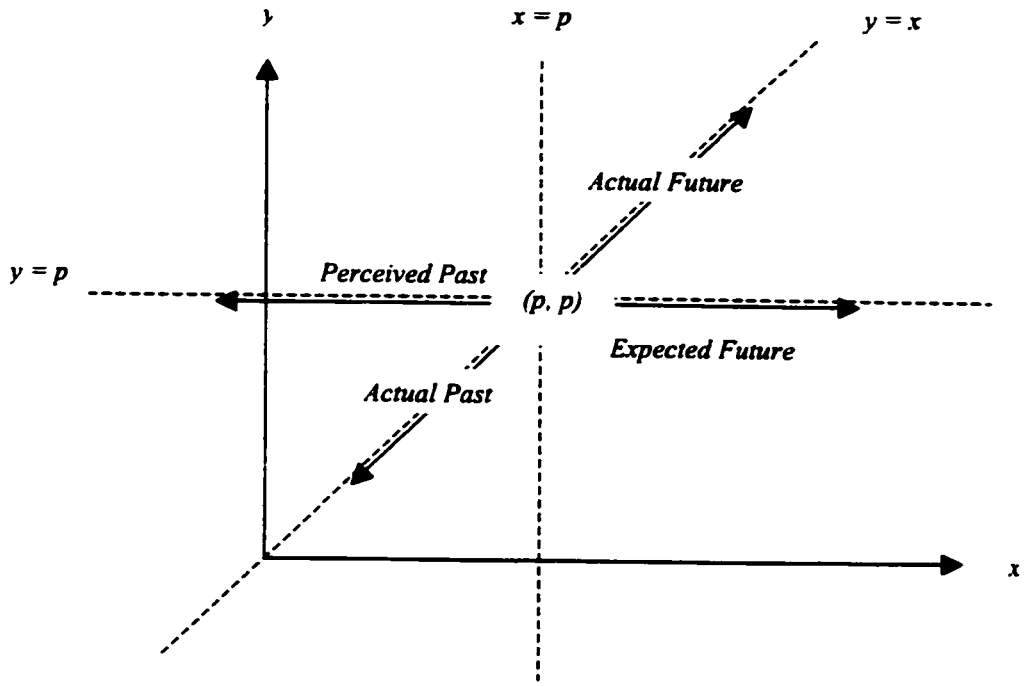

Figure 3.2 Trudel's model

Every predicate in the logic has two temporal arguments. The two temporal arguments do not specify an interval, but are Cartesian coordinates. For example, the predicate *white(5, 10)* specifies that *white* is true at the point *(5,10)* on the Cartesian plane.

Using this two-dimensional temporal structure, one has to consider the problem of two-dimensional persistence as opposed to traditional linear temporal structures that deal with persistence along a single axis only.

Chapter 4
Temporal Extension of FCA (TFCA)

4.1 Motivation

Motivated by different practical problems of visual input processing, formal knowledge representation, and recognition of continuous speech, an extension to FCA is proposed. The essence of this extension is that time plays a central role in almost every aspect of artificial intelligence ([Sho85], [Sho87] and [Sho88]). Concept lattices are good structures for representing the flow of temporal knowledge and representing temporal evolutions.

Understanding concept evolutions can be useful in many applications including data mining, planning, and decision-support systems. The application that motivated this work in particular is identifying the evolution of a set of concepts and related vocabulary for speech recognition in the Speech Web [FC99] as a conversation evolves.

4.2 Representation Issues

In extending FCA over time, we prefer to index an object with a time variable, rather than having time itself as an attribute for that object. The conventions we assume are as follows:

- O_{ti} represents Object O at time ti, where t is the time variable and i is an integer, and read as 'Object O at time t i'.
- ti precedes tj if $i < j$ for any two integers i and j.
- Given all the time intervals $t_{i \in n}$, where i is an integer and n is the number of observations made on a particular object over time, they form a total order.

Note that indexing an object has many advantages over representing time as an attribute in any given context. First, if the attributes of an object *O* change from time *ti* to time *tj*, then the change of attributes will be obvious in the context relation due to the different crosses relating object *O* to its attributes at times *ti* and *tj*. Second, it allows a flexible way of binding an object *O* to the time variable, and identifies the time *ti* at which a change in the attributes happens. As an example, refer to Figure 4.1.

Objects/Attributes	*a1*	*a2*
$(O1)_{ti}$	X	X
$(O1)_{tj}$	X	
$(O2)_{ti}$	X	X
$(O2)_{tj}$		X

Table 4.1 Notation representation

Third, it should be noted that making the time variable an attribute would impose an additional overhead dimension in the notation of FCA to the attributes in order to know what attribute an object *O* had at time *ti*, and would result in more blank cells in other examples where we have bigger contexts. Therefore, following this pattern will lead something similar to what is illustrated in Figure 4.2 in order to have the same information depicted in Figure 4.1.

	ti	*tj*	*ti*	*tj*
Objects/Attributes	*a1*	*a1*	*a2*	*a2*
O1	X	X	X	
O2	X		X	X

Table 4.2 Alternative representation

Finally, we advocate the approach depicted in Figure 4.1 by saying that interpreting the same object, *O1* for example, over two different points in time *ti* and *tj* as two different objects in two consecutive rows, is a philosophical question that needs to be addressed. We claim that objects $(O1)_{ti}$ and $(O1)_{tj}$ are two different objects as obviously the

intension of object *O1* changes from time *ti* to time *tj*, assuming that *ti* precedes *tj*, as it is appears in Figure 4.1.

The time model used in the temporal extension of FCA presented in this monograph is assumed to be similar to McDermott's model presented in Section 3.2.2. Examining the structure of the temporal formal context presented in Table 4.3, we can see that the time variables used to index objects form a total order for any particular observed object. However, they can also form a partial order among all the observed objects since, for example, time *t1* at which *Adam* is observed need not be the same time *t1* at which *Steve* is observed. Therefore, the time model we assume we are using needs to accommodate more than just a linear representation of temporal points or intervals presented by Allen in Section 3.2.1 that only orders them totally. A time model similar to McDermott's or even Trudel's presented in Sections 3.2.2 and 3.2.4 respectively will fit the purpose of our temporal extension of FCA.

4.3 Temporal Evolutions

Concepts change and evolve with time. In fact, change seems to be constant in a continuously changing world. In many domains such as science, medicine, finance, demographics, and weather patterns, change is noticeable from one time to another. The extension of concepts (set of objects) and their intensions (set of related attributes) may change, affecting how the entities are related. As a consequence, the concept lattice characterizing the relationships among a set of entities (objects and attributes) evolves over time. Temporal concept lattice metamorphosis is the change in the concept lattice over time. Finding the order of the changing attributes therefore defines the evolution itself in the concept lattice.

4.4 Advantages

In discovering useful patterns from a database researchers are increasingly relying on data visualization to complement data mining in the knowledge-discovery process.

Visualization helps developing insights and deduces the hidden regularities in the data. Animation seems to provide proper visualization for temporal evolution. However such animations can be easily generated from the proposed lattices. Moreover, the concept hierarchies provide a new tool for the study of the relationships between an interval and its subintervals.

Adding further notations to FCA's representation is important to extend it over time. The suggested extension opens up new areas of applications to FCA such as representing the course of infection for a disease, the life cycle of a software project, the evolution of social, economic, and population trends.

4.5 Temporal Lattices

4.5.1 Types of Edges

To handle the evolution phenomenon of concept structures and analyze temporal metamorphosis, a temporal extension of FCA is developed. This temporal extension involves the study of persistence, and other temporal properties implied by the data and concept lattices.

As an example, consider the simple database in Table 4.3, where some attributes change over time (juvenile, adult, senior) while others persist (dead). It can be easily seen from the concept lattice how the concepts manifest a change over different times (Figure 4.1). Recall that in our notation, t represents the time variable and t_i precedes t_j if $i<j$ for any two integers i and j. In addition, t_i and t_j form a total order.

To represent temporal evolutions in a concept lattice, we use two types of edges: temporal edges and non-temporal edges. The temporal edges allow the evolution of a particular object to be followed over time. A temporal precedence relation "<" is defined over time points. The direction of the arrow indicates this precedence. Non-

temporal edges are undirected, as they are not governed by the temporal precedence. In fact, non-temporal edges describe a concept at a particular point in time.

The temporal concept lattice in Figure 4.1 shows that there are transient attribute and persistent ones. For example, juvenile is a transient attribute as a juvenile becomes an adult but dead is a persistent attribute.

Objects/Attributes	Juvenile	Adult	Senior	Dead	Male	Female
$Adam_{t1}$	X				X	
$Adam_{t2}$		X			X	
$Steve_{t1}$		X			X	
$Steve_{t2}$			X		X	
$Nancy_{t1}$			X			X
$Nancy_{t2}$				X		X
$Mary_{t1}$				X		X
$Mary_{t2}$				X		X

Table 4.3 Temporal human context database

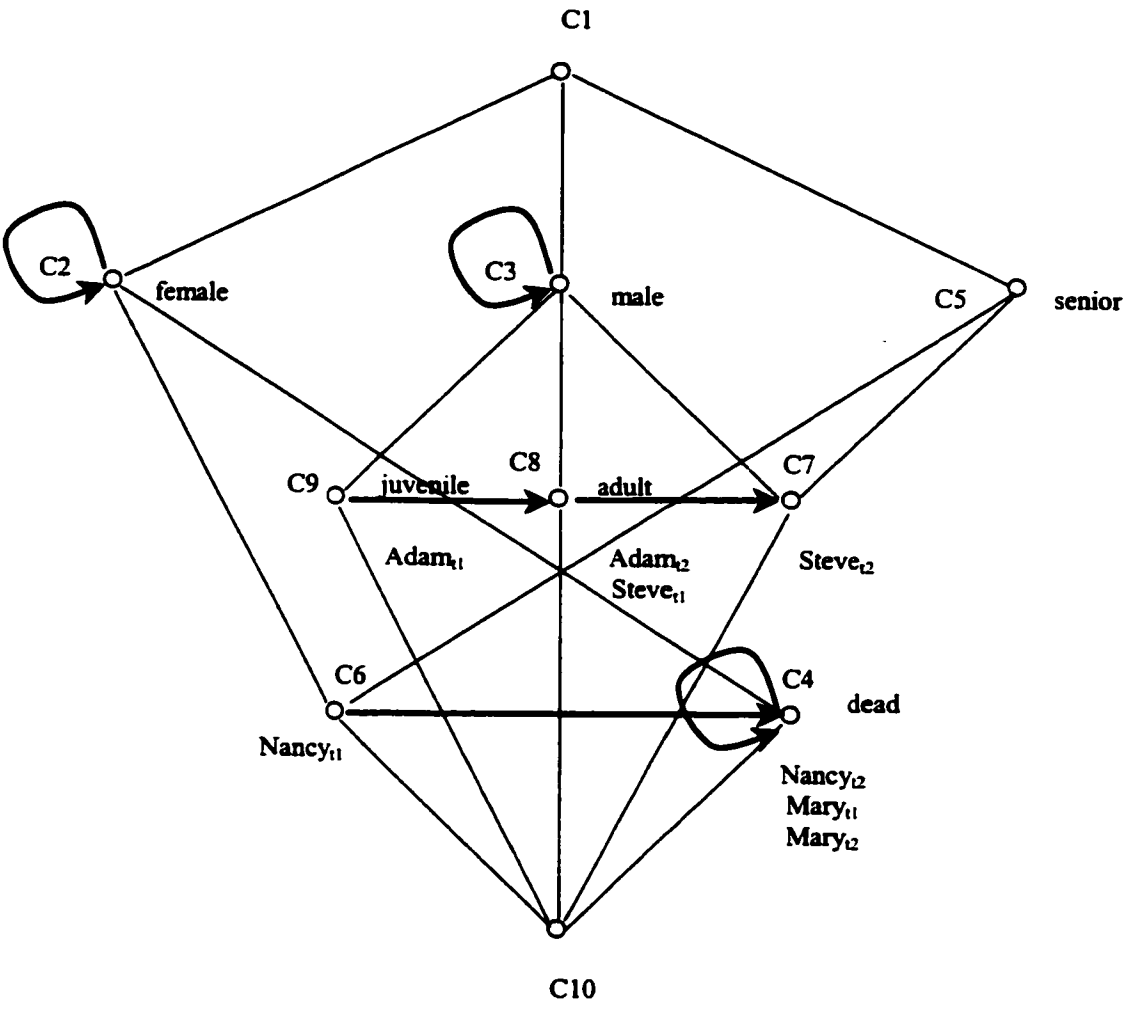

Figure 4.1 Temporal human context concept lattice

4.5.2 Precedence Relation

In order to extend formal concept analysis to represent temporal evolutions, we aim at encoding temporal precedence into formal concept analysis. A temporal precedence relation "<" is defined over time points, t_i, at which particular objects in the time-stamped database are being observed. This precedence relation is mainly represented by the direction of the temporal arrows shown in Figure 4.1.

4.6 Classification of Temporal Patterns

Patterns occur frequently in our daily life. A person, usually, eats three meals and sleeps at least one time a day. A crop fruit evolves from being inedible to being edible and then to a thrown away. Once a student gets a final course grade, that grade will persist on his transcripts. A person might own a car and then a bike, or a bike and then a car. Given all these examples, we classify temporal patterns into four categories: unconditional evolution patterns, conditional evolution patterns, persistence and transitions.

4.6.1 Unconditional Evolution Patterns

Unconditional patterns are any kind of a change in the attributes of an object over time that always happens in one unique direction. For example, during the process of getting older in humans, a person always evolves from being a child to an adult and then to a senior.

4.6.2 Conditional Evolution Patterns

Conditional patterns are any kind of change in the attributes of an object over time that might happen in more than one direction depending on a certain condition that controls the order of the changing attributes. For example, depending on the external temperature, a piece of ice might evolve from being solid to liquid or vice versa. Note that both conditional and unconditional patterns are forms of evolution and that some unconditional patterns become conditional once the data set has enough information about causes and effects.

4.6.3 Transitions

As opposed to evolution patterns, transitions are any kind of change in the attributes of an object over time that does not follow any specific direction. For example, a

person might eat then sleep, or sleep then eat. In this case, we say that eating and sleeping are transitions. Note that transitions do not constitute evolution patterns for our purpose here.

4.6.4 Persistence

Persistence is a state in which an object maintains its *acquired* attributes throughout a period of time without any change in these attributes once they are acquired. If that period of time covers the object's lifetime, we talk about an *absolute persistence*. Otherwise, it is a *relative persistence*. For example, if a person is born male and remains male until death, we say that *male* is an absolute persistent attribute. However, if after a period of time that person decides to change his sex then *male* is a relatively persistent attribute.

The term *acquired* is emphasized here since we cannot determine if a property is persistent unless an individual in the database has acquired it, and that it is persistent if it is once acquired it never changes. Note that *relative persistence* can form a stage of either an evolution pattern or a transient trend. Throughout this monograph, we use the term *persistence* to imply only *absolute persistence*.

4.7 Inferring Temporal Properties

Let B be the set of formal attributes for an object *Obj* at time t_i, and b_j be the attribute number j, where i and j are integers.

4.7.1 Intension of an Object

The intension of an object *Obj* at time t_i is the set of all attributes of that particular object at time t_i. It is written as:

$$i\{Obj_{ti}\} = \{b_j \in B\}. \qquad (1)$$

4.7.2 Evolution of an Object

An evolution of an object, *Ev(Obj)*, is an ordered set of sets containing all the intensions of this particular object from an arbitrary initial time t_0 to a certain time t_i. The intension of the object at time t_{i+1} is added to the resulting evolution set when we consider the evolution up to time t_{i+1}:

$$\forall t_0<t<t_i,\ i\{Obj_{ti}\} \in Ev_{t0\text{-}ti}(Obj)\ . \qquad (2)$$

For example, according to Table 1 above, we describe the evolution of *Adam* to be:

$$Ev_{t0\text{-}t2}(Adam) = \{\{male, juvenile\}, \{male, adult\}\}\ . \qquad (3)$$

4.7.3 Evolution Interval of an Object

The evolution interval (the subscript of the evolution) of an object, $Ev_{t0\text{-}ti}(Obj)$, specifies an interval from time t_0 to time t_i, where t_0 is the time at which the observation of the object started and t_i is the time at which the observation ended. In other words, the evolution depends on the interval we allow for the object to change. For example, according to Table 1, we describe the evolution of *Adam* at time t_1 to be:

$$Ev_{t0\text{-}t1}(Adam) = \{\{male, juvenile\}\}\ . \qquad (4)$$

4.7.4 Evolution Pattern of an Object

An evolution pattern of an object, *EvPat(Obj)*, is an ordered set containing the elements of the conditional or unconditional patterns that this particular object might exhibit throughout its lifetime. For example, according to Table 1, we describe the evolution pattern of *Adam* to be:

$$EvPat(Adam) = \{juvenile, adult\}\ . \qquad (5)$$

One way to determine the evolution pattern is to examine the intensions of a particular object over different points in time. For example, suppose that the intension

of *Ann* includes at time t_1 the *child* attribute, at time t_2 the *adult* attribute, and at time t_3 the *senior* attribute, such that $t_1 < t_2 < t_3$ and $<$ is a precedence relation that defines a temporal order. That is

$$i\{Ann_{t1}\} \supset \{juvenile\} . \qquad (6)$$
$$i\{Ann_{t2}\} \supset \{adult\} .$$
$$i\{Ann_{t3}\} \supset \{senior\} .$$

Then an evolution pattern from *juvenile* to *adult* to *senior* exists. Usually we are interested in evolution patterns that are consistent for all the objects or for a class of objects. For example, the pattern of evolution from *juvenile* to *adult* to *senior* is applicable to any object x of type *person* in the database. This evolution is written as:

$$\forall person(x \in G), \exists EvPat(x) \supseteq \{juvenile, adult, senior\} . \qquad (7)$$

To determine if such a pattern holds, we form a temporal matching problem [TS01]. Temporal matching can be formulated as a special constraint satisfaction problem that tries to find a consistent assignment of pattern times to observation/clock times consistent with states such as *juvenile*, *adult*, and *senior* to individuals whose state is only known at particular points in time. This assignment has to be consistent with temporal constraints representing the temporal pattern.

4.7.5 Transient Properties of an Object

A transient property of an object, *Transient(Obj)*, is a set containing all that object's attributes that exhibited a change in all of the intension sets of that object over time, including its evolution time if that object happened to have an evolution.

Note that an evolution pattern for an individual or a subclass may be a transient property for the class. In other words, a change that happens in any order is not considered to be an evolution pattern for a class, even though it is may be an evolution pattern for an individual. For example, suppose that the intension of *Bob* includes at time t_1 the *eating* attribute, at time t_2 the *sleeping* attribute while the

intension of *Sam* includes at time t_1 the *sleeping* attribute, at time t_2 the *eating* attribute. That is:

$$i\{Bob_{t1}\} \supset \{eating\} . \quad (8)$$
$$i\{Bob_{t2}\} \supset \{sleeping\} .$$
$$i\{Sam_{t1}\} \supset \{sleeping\} .$$
$$i\{Sam_{t2}\} \supset \{eating\} .$$

This pattern for *Bob* and *Sam* may be an evolution pattern for these two individuals if the observations indicate that at all observation times *Bob* eats before he sleeps, while *Sam* sleeps before he eats. In this case, transition is only a class property:

$$EvPat(Bob) = \{eating, sleeping\} . \quad (9)$$
$$EvPat(Sam) = \{sleeping, eating\} .$$
$$\forall x\ (person(x \in G)),\ Transient(x) \supseteq \{eating, sleeping\} .$$

However, if the observations indicate that sometimes *Bob* eats after he sleeps, while *Sam* sleeps after he eats, then transition in this case is both an individual and a class property:

$$Transient(Bob) = \{eating, sleeping\} . \quad (10)$$
$$Transient(Sam) = \{sleeping, eating\} .$$
$$\forall x\ (person(x \in G)),\ Transient(x) \supseteq \{eating, sleeping\} .$$

Note that a transient property is not an ordered set, and therefore the following transients of a class x are equivalent:

$$Transient(x) = \{eating, sleeping\} . \quad (11)$$
$$Transient(x) = \{sleeping, eating\} .$$

Note also that the temporal matching exercise detects such transients as failure to match [TS01].

4.7.6 Persistent Properties of an Object

A persistent property of an object, *Persist(Obj)*, is a set containing all that object's attributes that persisted in all of the intension sets of that object over time, including its evolution time if that object happened to have an evolution. For example, according to Table 1, we describe the persistent property of *Adam* to be:

$$Persist(Adam) = \{male\} . \qquad (12)$$

Note that we cannot determine if a property is persistent if no individual in the database has acquired it, and that it is persistent if it is once acquired it never changes. One way to determine a persistent property of an object is to examine its intensions over different points in time. For example, if the intension of *John* includes at times t_1, t_2 and t_3 the *male* attribute then *male* is a persistent property of *John* in particular, or is an individual persisting property:

$$i\{John_{t1}\} \supset \{male\} . \qquad (13)$$

$$i\{John_{t2}\} \supset \{male\} .$$

$$i\{Bill_{t1}\} \supset \{male\} .$$

$$i\{Bill_{t2}\} \supset \{male\} .$$

$$Persist(John) \supseteq \{male\} . \qquad (14)$$

$$Persist(Bill) \supseteq \{male\} .$$

In this case, we may have persistence as a class property when all objects x of type *person* in the database who are *male* have the *male* property persisting:

$$\forall x\ (person(x \in G) \wedge male(x)),\ Persist(x) \supseteq \{male\} . \qquad (15)$$

Similarly, if the *female* property is persistent to all objects y of type *person* in the database who are *female*, then *female* is a class persisting property:

$$\forall x\ (person(x \in G) \wedge female(x)),\ i\{x\} \supseteq \{female\} . \qquad (16)$$

Note that allowing multi-valued attributes enables us to consider *gender* or *eye color*, for example, as persistent properties.

A persistent property for an individual or a subclass, however, may be a transient property for the class. For example, if the observations indicate that at a later time *John* has undergone a transsexual operation and now possesses the *female* attribute, while *Bill* still possesses the male attribute, then *male* is still an individual persisting property for *Bill* but a transient property for both *John* and the class x of male persons:

$$i\{John_{t3}\} \supset \{female\} . \qquad (17)$$

$$i\{Bill_{t4}\} \supset \{male\} .$$

$$Transient(John) \supseteq \{male, female\} . \qquad (18)$$

$$Persist(Bill) \supseteq \{male\} .$$

Chapter 5
STEP and TLAT Algorithms

5.1 Inferring Evolution Patterns

Given the sequence of all the objects' attributes in the temporal database, the problem of finding and inferring the temporal properties is reduced to the problem of mining sequential patterns where the patterns are the attributes of the objects observed at different times. Finding persistent properties of an object is straightforward. Differentiating between evolution patterns and transient properties, however, requires much more effort. Having the user specify a certain support threshold that indicates the fraction of all the objects that support a specific attribute sequence usually helps in differentiating between evolution patterns and transient properties.

5.1.1 Mining Sequential Patterns

The problem of mining sequential patterns is the problem of finding the maximal sequences in the database among all sequences that have a user-specified minimum support. Each such maximal sequence represents a sequential pattern ([AS95], [SA96], and [Ram98]). Sequential pattern discovery can be thought of as association-rule discovery over a temporal database [Zak97], and as a discovery of inter-transaction patterns/inter-attribute patterns (sequences/intensions) rather than a discovery of intra-transaction patterns/intra-attribute patterns (itemsets/attributes).

An efficient algorithm for mining sequential patterns is called GSP - *Generalized Sequential Patterns* [SA96]. GSP makes multiple passes over the databases, where every subsequent pass starts with a seed set, the frequent sequences found in the previous pass, and uses it to generate new potentially longer frequent sequences. Zaki ([Zak97] and [Zak01]) presents an algorithm for fast discovery of sequential patterns called SPADE -

Sequential PAttern Discovery using Equivalence classes. SPADE finds all frequent sequences in only three database scans, outperforms GSP by more than a factor of two and by an order of magnitude with some pre-processed data, and has linear scalability with respect to the number of input sequences [Zak01].

5.2 STEP: A New Algorithm for Inferring Temporal Properties

In what follows we describe STEP – *Sequential TEmporal Properties*, the new algorithm for inferring temporal properties. STEP requires a mapping of the SPADE algorithm to the problem domain in hand. SPADE ([Zak97] and [Zak01]) is an algorithm for fast discovery of sequential patterns whose applications are in the retail sales or market-basket analysis domains. Given a collection of items, a set of records over those items, and records belonging to the customer, the task is to identify all the commonly occurring sequences of items bought by the customers.

In this case, the customers would map to the objects being observed in the time-stamped database, the items the customer buys would map to the attributes that an objects possesses, the transaction or set of items that a customers buys at one time would map to the intension or set of attributes that a particular object has at one time, and the transaction identifier would map to the time (t_i) at which that object is observed. Note that an object might have more than one attribute at a time just like a customer might buy more than one object at a time.

STEP, however, does not only consist of using the mapped version of SPADE, MSPADE. Using MSPADE only forms the first phase of the algorithm. The second phase will consist of a temporal matching module that can be used to fit the individuals whose patterns are missing to the frequent sequences obtained in the mapped version of the SPADE algorithm. The third phase consists of the TLAT, *Temporal LATtices*, algorithm that adds temporal edges to the temporal lattice corresponding to the given temporal context. The complete phases of the algorithm can be seen in Figure 5.1.

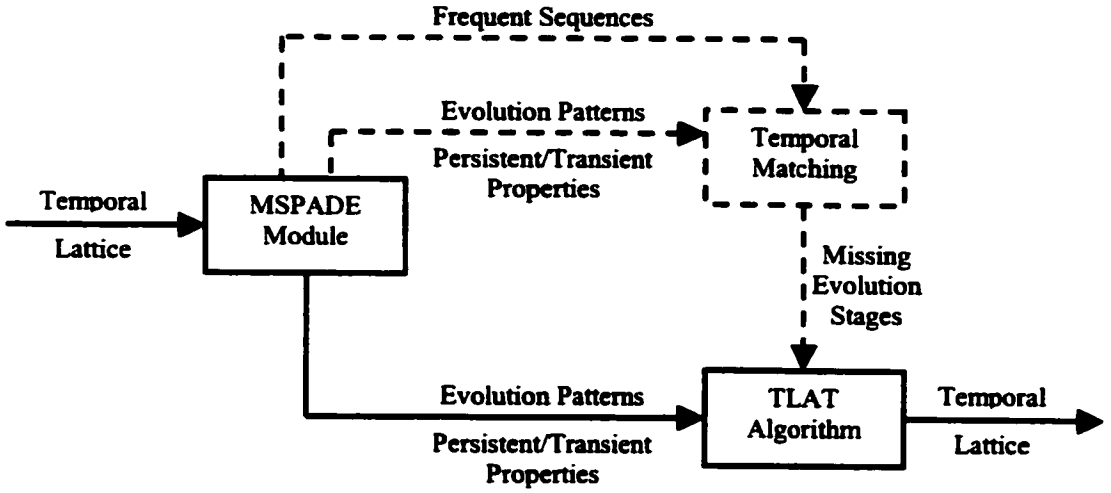

Figure 5.1 Different phases of STEP algorithm

MSAPDE takes the time-stamped database and the user-specified minimum support as input, and, according to this support, extracts the set of evolution patterns, persistent and transient properties along with the set of frequent sequences from the temporal database. Two scenarios are then possible to happen. In the first scenario, MSPADE outputs these two sets together and feeds them as input to the second phase consisting of the temporal matching module which, in turn, outputs the set of missing evolution stages for all individuals who do not show complete evolution stages and feeds them as input to the third phase consisting of the TLAT algorithm. In the second scenario, MSPADE feeds only the first set of evolution patterns, persistent and transient properties alone to the third phase directly. Finally, the TLAT algorithm takes its input and works on generating the temporal edges of the temporal lattice corresponding to the user-specified minimum support.

Note that in this monograph, we only describe and work on implementing the second scenario, and leave the development of the temporal matching module as part of work that shall be conducted in the future.

5.2.1 STEP Design

The STEP algorithm consists of four main classes:

- One class *TplObj* standing for a temporal object
- Three others that extend *TplObj*: *Patterns*, *LCSObj*, and *LMSObj* standing for evolution and persistent/transient patterns, longest common subsequence object, and longest monotonic subsequence respectively.

The complete design of STEP class hierarchies is shown in Figure 5.2. We describe what each method is doing in pseudo code in Section 5.2.2. The structure of each data structure used is shown in Section 5.2.3 where we present a case study for the algorithm. Note also that initially STEP reads from a file called *patterns.idx* information regarding the name of each object and how many times that particular object is observed. Then after reading each line of that file, it goes to read from a random access file, *patterns.dat*, detailed information about the object being observed at different times. Finally, STEP outputs its results, the evolution patterns and persistent/transient properties, onto a file called *temprop.dat*.

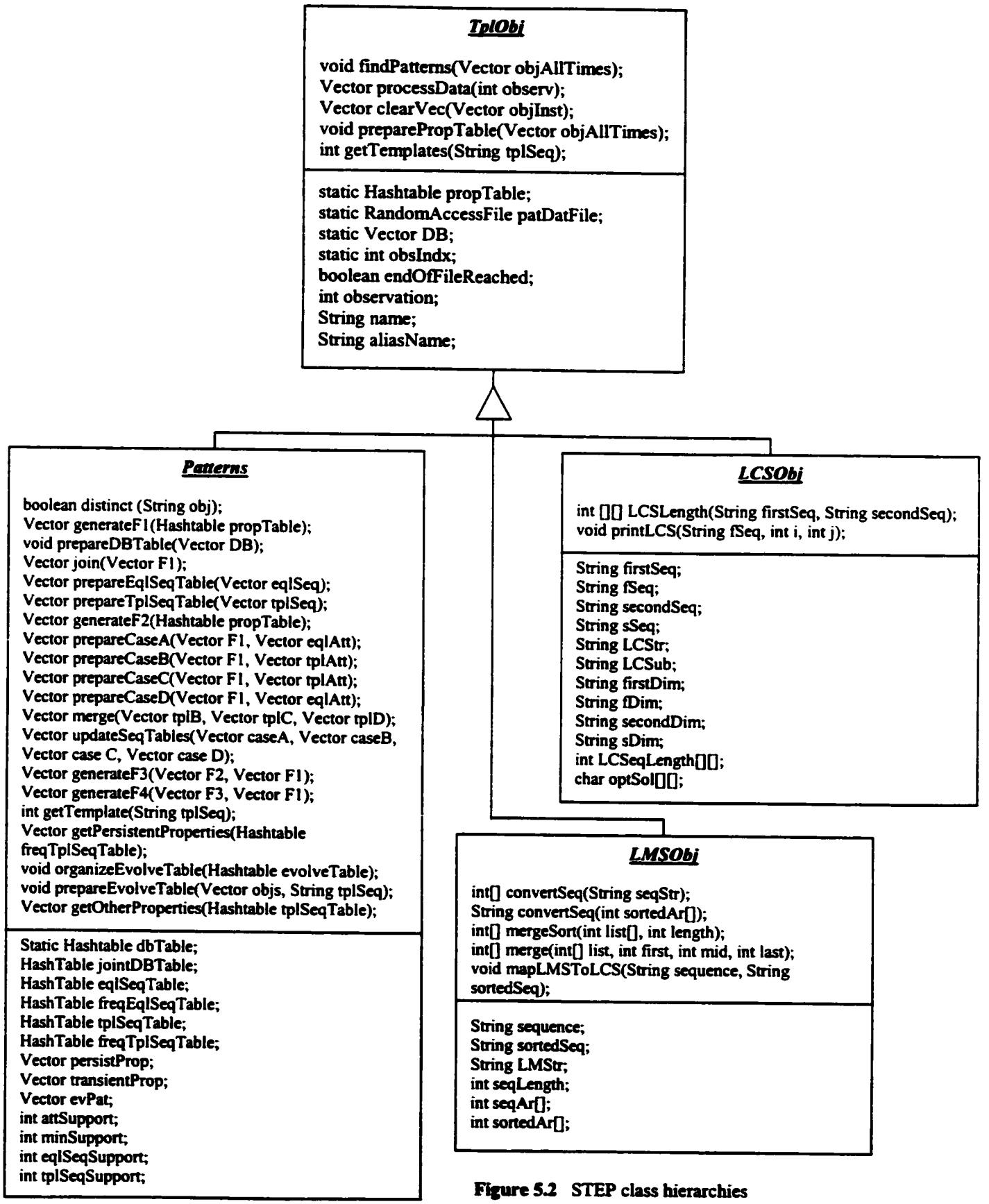

Figure 5.2 STEP class hierarchies

5.2.2 STEP Pseudo Code

```
//-------------------------------------------------------
//- Scan temporal database
//-------------------------------------------------------
for each object in 'patterns.idx' file
{
    index = number of observations;
    objAllTimes = obj.processData(index);
    obj.preparePropTable(objAllTimes);
}
//-------------------------------------------------------
//- Generate frequent sequences
//-------------------------------------------------------
F1 = generateF1(propTable);
F2 = generateF2(F1);
F3 = generateF3(F2, F1);
F4 = generateF4(F3, F1);
//-------------------------------------------------------
//- Determine persistent proprerties
//-------------------------------------------------------
persistProp = getPersistentProperties(freqTplSeqTable);
//-------------------------------------------------------
//- Determine evolution patterns and transient properties
//-------------------------------------------------------
otherProp = getOtherProperties(tplSeqTable);
evPat = (Vector)otherProp.elementAt(0);
transientProp = (Vector)otherProp.elementAt(1);
//-------------------------------------------------------
//- Output evolution patterns, persistent and transient
//- properties onto 'temprop.dat'
//-------------------------------------------------------
output.writeBytes(persistProp.toString() + "\n");
output.writeBytes(evPat.toString() + "\n");
output.writeBytes(transientProp.toString() + "\n");
```

Figure 5.3 The STEP algorithm

We will describe in pseudo code the major modules used in STEP. These modules are *processData(int index)* and *preparePropTable(Vector objAllTimes)* in the main control, and *generateF1(Hashtable PropTable)*, *generateF2(Vector F1)*, *generateF3(F2, F1)*, *getPersistentProperties(Hashtable freqTplSeqTable)*, and *getOtherPoperties(Hashtable tplSeqTable)* in *Patterns.java* class. Note that the code for these modules can be found in *Appendix A* at the end of this monograph.

```
for each observed instance of the object objInst in 'patterns.dat'
{
    objAllTimes.addElement(objInst);
    dbObj = clearVec(objInst);
    DB.addElement(dbObj);
}
return objAllTimes;
```

Figure 5.4 processData() module

```
for each observed instance of the object objInst in objAllTimes
{
    clrObjInst = clearVec(objInst);
    for each attribute att in clrObjInst
    {
        objID = all objects having attribute att along with their IDs;
        propTable.put(att, objID);
    }
}
```

Figure 5.5 prepareProptable() module

```
for each attribute att in propTable
{
    objIDs = (Vector)propTable.get(att);
    for each distinct object obj in objIDs
        attSupport++;
    if (attSupport >= minSupport)
        result.addElement(att);
}
return result;
```

Figure 5.6 generateF1() module

```
prepareDBTable(DB);
seqList = join(F1);
eqlSeq = (Vector)seqList.elementAt(0);
tplSeq = (Vector)seqList.elementAt(1);
eqlSup = prepareEqlSeqTable(eqlSeq);
tplSup = prepareTplSeqTable(tplSeq);
result.addElement(eqlSup);
result.addElement(tplSup);
return result;
```

Figure 5.7 generateF2() module

```
for each dbObj in DB
{
    //-----------------------------------------------------
    //- Prepare 'db' hash table
    //-----------------------------------------------------
    objAtts = object attributes at one observation;
    dbTable.put(objName, objAtts);
    //-----------------------------------------------------
    //- Prepare 'jointDBTable' hash table
    //-----------------------------------------------------
    tplAtts = all object attributes at all observations;
    jointDBTable.put(aliasName, tplAtts);
}
```

Figure 5.8 prepareDBTable() module

```
for each attribute att in F1
{
    temp1 = (Vector){F1.clone() - att};
    temp2 = (Vector)F1.clone();
    for each attribute att1 in temp1
        eqljoin.addElement(att + " " + att1);
    for each attribute att2 in temp2
        eqljoin.addElement(att + "->" + att2);
}
result.addElement(eqljoin);
result.addElement(tplJoin);
return result;
```

Figure 5.9 join() module

```
for each equality sequence instance eqlSeqInst in eqlSeq
{
    //-------------------------------------------------
    //- Prepare 'eqlSeqTable' hash table
    //-------------------------------------------------
    for each object obj in dbTable
    {
        if obj has eqlSeqInst and obj is distinct
            eqlSupport++;
        objList = all objects in dbTable having eqlSeqInst;
        eqlSeqTable.addElement(eqlSeqInst, objList);
    }
    //-------------------------------------------------
    //- Prepare 'freqEqlSeqTable' hash table
    //-------------------------------------------------
    if (eqlSupport >= minSupport)
    {
        freqEqlSeqTable.put(eqlSeqInst, objList);
        result.addElement(eqlSeqInst);
        result.addElement(eqlSeqSupport);
    }
}
return result;
```

Figure 5.10 prepareEqlSeqTable() module

```
for each temporal sequence instance tplSeqInst in tplSeq
{
    //-------------------------------------------------
    //- Prepare 'tplSeqTable' hash table
    //-------------------------------------------------
    for each object obj in jointDBTable
    {
        if obj has each single attribute in tplSeqInst at different observation times
            tplSupport++;
        objList = all objects in jointDBTable having tplSeqInst attributes at different times;
        eqlSeqTable.addElement(tplSeqInst, objList);
    }
    //-------------------------------------------------
    //- Prepare 'freqTplSeqTable' hash table
    //-------------------------------------------------
    if (tplSupport >= minSupport)
    {
        freqTplSeqTable.put(tplSeqInst, objList);
        result.addElement(tplSeqInst);
        result.addElement(tplSeqSupport);
    }
}
return result;
```

Figure 5.11 prepareTplSeqTable() module

```
eqlAtt = (Vector)F2.elementAt(0);
tplAtt = (Vector)F2.elementAt(1);
if ( !eqlAtt.isEmpty() )
{
    caseA = prepareCaseA(F1, eqlAtt);
    caseD = prepareCaseD(F1, eqlAtt);
}
if ( !tplAtt.isEmpty() )
{
    caseB = prepareCaseB(F1, tplAtt);
    caseC = prepareCaseC(F1, tplAtt);
}
result = updateSeqTables(caseA, caseB, caseC, caseD);
return result;
```

Figure 5.12 generateF3() module

```
for each sequence Seq in eqlAtt
{
    for each frequent attribute att in F1
    {
        if last attribute in Seq is not equal to att
            result.addElement(Seq + " " + att);
    }
}
return result;
```

Figure 5.13 prepareCaseA() module

```
for each sequence Seq in tplAtt
{
    for each frequent attribute att in F1
    {
        if last attribute in Seq is not equal to att
            result.addElement(Seq + " " + att);
    }
}
return result;
```

Figure 5.14 prepareCaseB() module

```
for each sequence Seq in tplAtt
{
    for each frequent attribute att in F1
            result.addElement(Seq + "->" + att);
}
return result;
```

Figure 5.15 prepareCaseC() module

```
for each sequence Seq in eqlAtt
{
    for each frequent attribute att in F1
            result.addElement(Seq + "->" + att);
}
return result;
```

Figure 5.16 prepareCaseD() module

```
eqlSup = prepareEqlSeqTable(caseA);
tplB = prepareTplSeqTable(caseB);
tplC = prepareTplSeqTable(caseC);
tplD = prepareTplSeqTable(caseD);
tplSup = merge(tplB, tplC, tplD);
result.addElement(eqlSup);
result.addElement(tplSup);
return result;
```

Figure 5.17 updateSeqTables() module

```
if ( !tplB.isEmpty() )
{
    for each frequent temporal sequence freqTplSeq in tplB
        result.addElement(freqTplSeq);
}
if ( !tplC.isEmpty() )
{
    for each frequent temporal sequence freqTplSeq in tplC
        result.addElement(freqTplSeq);
}
if ( !tplD.isEmpty() )
{
    for each frequent temporal sequence freqTplSeq in tplD
        result.addElement(freqTplSeq);
}
return result;
```

Figure 5.18 merge() module

```
result = generateF3(F3, F1);
return result;
```

Figure 5.19 generateF4() module

```
for each temporal sequence tplSeq in freqTplSeqTable
{
    if tplSeq has the right template
    //-------------------------------------------------------
    //- all single attributes in tplSeq are the same,
    //- and no equality joined attributes are found
    //-------------------------------------------------------
    att = single attribute in tplSeq;
    objs = all objects in freqTplSeqTable having tplSeq;
    result.addElement(att);
    result.addElement(objs);
    persistProp.addElement(att);
}
return result;
```

Figure 5.20 getPersistentProperties() module

```
//-------------------------------------------------------
//- preparing 'evolve' hash table
//-------------------------------------------------------
for each evolving object obj in objs
{
    evolVec = all temporal evolution sequences for obj;
    evolveTable.put(obj, evolVec);
}
```

Figure 5.21 prepareEvolveTable() module

```
for each evolving object obj in evolveTable
{
    //-------------------------------------------------------------
    //- organizing 'evolve' hash table 'values'
    //- to make transitions easier to detect
    //-------------------------------------------------------------
    evolVec = all temporal evolution sequences for obj;
    replace all temporal sequences in evolVec that form transitive links;
    take out duplicates from evolVec;
    //-------------------------------------------------------------
    //- restoring 'evolve' hash table values
    //-------------------------------------------------------------
    newEvol = (Vector)evolVec.clone();
    evolveTable.remove(obj);
    //-------------------------------------------------------------
    //- check template of new evolution sequences
    //- remove objects from evolveTable who have ALL
    //- sequences of the persisting template A->A
    //-------------------------------------------------------------
    for each evolution sequence evSeq in newEvol
    {
        if ( getTemplates(evSeq) == 0 )
            newEvol.removeElement(evSeq);
    }
    if ( newEvol.size() != 0 )
        evolveTable.put(obj, newEvol);
}
```

Figure 5.22 organizeEvolveTable() module

```
for each temporal sequence tplSeq in tplSeqTable
{
    if tplSeq has the right template
    //-------------------------------------------------------
    //- all single attributes in tplSeq are different,
    //- and no equality joined attributes are found
    //-------------------------------------------------------
    objs = all objects in freqTplSeqTable having tplSeq;
    if ( objs.size() >= minSupport )
        prepareEvolveTable(objs, tplSeq);
    else
    {
        if ( (objs.size() < minSupport) && (objs.size() > 0) )
        {
            transition.addElement(objs);
            transition.addElement(tplSeq);
            transientProp.addElement(tplSeq);
        }
    }
}
organizeEvolveTable(evolveTable);
for each evolving object obj in evolveTable
{
    evolVec = all temporal evolution sequences for obj;
    evolution.addElement(obj);
    evolution.addElement(evolVec);
    evPat.addElement(evolVec);
}
result.addElement(evolution);
result.addElement(transition);
return result;
```

Figure 5.23 getOtherProperties() module

5.2.3 A Case Study

In what follows we describe a case study of the STEP algorithm, and restrict ourselves to show the results of the simple temporal context presented in Table 4.3. We first show the contents of the two input files, *patterns.idx* and *patterns.dat*, and the output file, *temprop.dat*, corresponding to that context in Figures 5.24, 5.25, and 5.26 respectively. Then, we present the structure of the different populated index tables used when the algorithm is run with a support set to 1 in Tables 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, and 5.10. We also show the output that the STEP algorithm generates for the same support in Table 5.11.

```
adam    2
steve   2
nancy   2
mary    2
end     99
```

Figure 5.24 'patterns.idx' file

adamt1	juvenile	null	null	null	male	null
adamt2	null	adult	null	null	male	null
stevet1	null	adult	null	null	male	null
stevet2	null	null	senior	null	male	null
nancyt1	null	null	senior	null	null	female
nancyt2	null	null	null	dead	null	female
maryt1	null	null	null	dead	null	female
maryt2	null	null	null	dead	null	female

Figure 5.25 'patterns.dat' file

```
Persistent Properties:
[male, [steve, adam], dead, [mary], female, [mary, nancy]]

Evolution Patterns:
[mary, [dead->dead, female->female], nancy, [senior->dead],
steve, [adult->senior], adam, [juvenile->adult]]

Transient Properties:
[ ]
```

Figure 5.26 'temprop.dat' file

Table 5.1 shows the frequent attribute sequences found in the temporal context for the given support. Note that the frequent 2-sequences are obtained by performing a self-join on the frequent attributes found in frequent 1-sequences. We therefore guarantee that all subsequences of a frequent sequence are frequent ([Zak97] and [Zak01]). This is very important for the sake of restricting the search for subsequent frequent sequences.

Frequent 1-Sequences	**Support**
senior	2
male	2
adult	2
juvenile	1
dead	2
female	2
Frequent 2-Sequences	**Support**
senior male	1
senior female	1
adult male	2
juvenile male	1
dead female	2
senior->dead	1
senior->female	1
male->senior	1
male->male	2
male->adult	1
adult->senior	1
adult->male	1
juvenile->male	1
juvenile->adult	1
dead->dead	1
dead->female	1
female->dead	2
female->female	2
Frequent 3-Sequences	**Support**
senior->dead female	1
male->senior male	1
male->adult male	1
adult->senior male	1
juvenile->adult male	1
dead->dead female	1
female->dead female	2
senior female->dead	1
senior female->female	1

adult male->senior	1
adult male->male	1
juvenile male->male	1
juvenile male->adult	1
dead female->dead	1
dead female->female	1
Frequent 4-Sequences	**Support**
senior female->dead female	1
adult male->senior male	1
juvenile male->adult male	1
dead female->dead female	1

Table 5.1 Frequent attribute sequences of Table 4.3 with support = 1.

Note also in frequent 2-sequences and frequent 3-sequences the differentiation between two types of join. The first one is an equality join that usually describes the occurrence of two or more attributes/items at the same time for a particular observed object/customer. The second is a temporal join that describes the occurrence of two or more attributes/items at different times for a particular observed object/customer ([Zak97] and [Zak01]). In our case, we separate equality joined attributes by a space while an arrow, "->", separates temporally joined attributes.

Table 5.2 presents an index table, *propTable*, that stores for each of the formal attributes found in the formal context a list of the objects having these attributes along with the times, t_i, during which these object have been observed to possess these attributes.

senior	[steve, t2, nancy, t1]
male	[adam, t1, adam, t2, steve, t1, steve, t2]
adult	[adam, t2, steve, t1]
juvenile	[adam, t1]
dead	[nancy, t2, mary, t1, mary, t2]
female	[nancy, t1, nancy, t2, mary, t1, mary, t2]

Table 5.2 'propTable' table

Adam$_{t1}$	juvenile	Male
Adam$_{t2}$	adult	Male
Steve$_{t1}$	adult	male
Steve$_{t2}$	senior	male
Nancy$_{t1}$	senior	female
Nancy$_{t2}$	dead	female
Mary$_{t1}$	dead	female
Mary$_{t2}$	dead	female

Table 5.3 'dbTable' table

Adam	juvenile male, adult male
Steve	adult male, senior male
Nancy	senior female, dead female
Mary	dead female, dead female

Table 5.4 'jointDBTable' table

Tables 5.3 and 5.4 are two different ways of representing the context presented in Table 4.3. Tables 5.5 and 5.6 describe the contents of the "equality sequential" and "frequent equality sequential" index tables respectively after the program has finished execution. Similarly, Tables 5.7 and 5.8 describe the contents of the "temporal sequential" and "frequent temporal sequential" index tables respectively. Note that

since the size of Table 5.7 is fairly big (381 key elements corresponding to the context shown in Table 4.3), we only show the values of 52 key elements of that table.

juvenile male female=[]	adult male adult=[]	senior adult=[]	senior female senior=[]
senior male adult=[]	juvenile adult=[]	adult female=[]	male dead=[]
senior female adult=[]	male female=[]	dead senior=[]	dead female male=[]
juvenile male=[adamt1]	senior male juvenile=[]	dead female dead=[]	female male=[]
juvenile female=[]	senior male female=[]	male juvenile=[]	senior dead=[]
adult male female=[]	senior female juvenile=[]	senior male dead=[]	adult male=[stevet1, adamt2]
adult male juvenile=[]	juvenile dead=[]	adult male senior=[]	adult male dead=[]
female juvenile=[]	juvenile male senior=[]	juvenile male adult=[]	juvenile male juvenile=[]
female dead=[]	dead female=[maryt2, maryt1, nancyt2]	senior female=[nancyt1]	senior juvenile=[]
juvenile senior=[]	senior male=[stevet2]	dead adult=[]	senior female male=[]
adult dead=[]	male senior=[]	adult senior=[]	dead male=[]
female senior=[]	male adult=[]	dead female juvenile=[]	dead female senior=[]
dead juvenile=[]	female adult=[]	senior female dead=[]	juvenile male dead=[]
dead female adult=[]	adult juvenile=[]	senior male senior=[]	

Table 5.5 'eqlSeqTable' table

dead female	[maryt2, maryt1, nancyt2]
senior female	[nancyt1]
juvenile male	[adamt1]
adult male	[stevet1, adamt2]
senior male	[stevet2]

Table 5.6 'freqEqlSeqTable' table

male->adult senior=[]	adult->male adult=[]	male->male->female=[]	senior female->dead adult=[]
juvenile male->adult->dead=[]	male->adult male dead=[]	adult male->male->senior=[]	dead->female->female=[]
juvenile male->senior=[]	juvenile male->adult->senior=[]	male->male->dead=[]	senior female->dead male=[]
dead->dead->juvenile=[]	adult male->juvenile=[]	senior->dead male=[]	juvenile male->male=[adam]
senior->female dead=[]	senior->female adult=[]	adult->senior->male=[]	male->senior male juvenile=[]
dead->female adult=[]	juvenile->adult senior=[]	juvenile->adult->senior=[]	adult male->male->dead=[]
Senior female->dead->female=[]	dead female->dead=[mary]	male->senior male=[steve]	female->juvenile=[]
juvenile male->male->adult=[]	senior female->female senior=[]	senior->juvenile=[]	senior female->dead senior=[]
juvenile->adult male->female=[]	female->female->juvenile=[]	male->male female=[]	senior->dead female juvenile=[]
dead female->female->senior=[]	senior female->dead female=[nancy]	male->adult male=[adam]	male->adult male adult=[]
dead->dead->male=[]	female->dead juvenile=[]	juvenile->adult male senior=[]	adult->senior male senior=[]
Female->dead female->male=[]	dead->dead female adult=[]	juvenile->adult->juvenile=[]	juvenile->juvenile=[]
Senior->female=[nancy]	dead->female juvenile=[]	dead->dead->senior=[]	senior female->female=[nancy]

Table 5.7 'tplSeqTable' table

senior->dead female	[nancy]
male->senior	[steve]
dead->female	[mary]
female->dead	[mary, nancy]
senior female->dead	[nancy]
dead female->female	[mary]
senior->dead	[nancy]
juvenile male->adult male	[adam]
adult male->senior male	[steve]
adult male->male	[steve]
adult->male	[steve]
dead female->dead female	[mary]
male->male	[steve, adam]
male->adult male	[adam]
senior female->female	[nancy]
adult male->senior	[steve]
juvenile male->male	[adam]
adult->senior male	[steve]
juvenile->adult	[adam]
dead->dead	[mary]
juvenile->male	[adam]
dead female->dead	[mary]
female->dead female	[mary, nancy]
juvenile->adult male	[adam]
male->adult	[adam]
adult->senior	[steve]
male->senior male	[steve]
juvenile male->adult	[adam]
dead->dead female	[mary]
senior->female	[nancy]
senior female->dead female	[nancy]
female->female	[mary, nancy]

Table 5.8 'freqTplSeqTable' table

Nancy	[senior->female, senior->dead, female->dead]
Adam	[male->adult, juvenile->male, juvenile->adult]
Mary	[dead->female, female->dead]
Steve	[male->senior, adult->senior, adult->male]

Table 5.9 Preliminary 'evolveTable' table

Nancy	[senior->dead]
Adam	[juvenile->adult]
Steve	[adult->senior]

Table 5.10 Final 'evolveTable' table

Finally, Tables 5.9 and 5.10 show a preliminary version and a final version of the *'evolveTable'* index table respectively. The final version replaces all transitive sequences found in the preliminary version. Table 5.11 shows the output that the STEP algorithm generates for the specified support. This table basically has the same contents as the *'temprop.dat'* file presented in Figure 5.26. This output will be fed as input to the TLAT algorithm, described in Section 5.7, which handles the responsibility of drawing the temporal edges for the given temporal context based on the support that the user specifies.

Persistent Properties	[male, [steve, adam], dead, [mary], female, [mary, nancy]]
Evolution Patterns	[nancy, [senior->dead], steve, [adult->senior], adam, [juvenile->adult]]
Transient Properties	[]

Table 5.11 Output of STEP algorithm for support = 1

5.3 STEP Limitations

At the present, the STEP algorithm does not make use of equivalence classes that help partitioning the problem of enumerating all frequent sequences, and performing temporal joins to obtain sequence supports ([Zak97] and [Zak01]). As a result of this, STEP does more work in terms of computation than SPADE. Equivalence classes are important since we only have limited amount of main memory, and as the size of the temporal database grows bigger the intermediate join tables produced for enumerating subsequent frequent sequences might not fit in memory. Therefore, equivalence classes address this problem by decomposing it into smaller pieces such that each piece can be solved independently in main memory. In addition, equivalence classes can play a fundamental role in driving the STEP algorithm to an end as soon as no new frequent classes are generated after computing the sets of frequent 1-sequences *F1*, and the set of frequent 2-sequences *F2* ([Zak97] and [Zak01]).

Another issue is that even though the design and implementation of the *LCSObj* and *LMSObj* (longest common subsequence and longest monotonic subsequence) classes are found in this documentation, we do not make use of them at the present in the STEP algorithm. The *LCSObj* and *LMSObj* classes were initially designed to identify the patterns of any given attribute sequence, and to determine how far an object is in an evolution once the evolution patterns in the time-stamped database have been extracted. Therefore, they are extremely useful for the design of a temporal matching module, as a second phase for the STEP algorithm described in Figure 5.1, which can be investigated and implemented in the future. All what the STEP algorithm does at present is having the user input a support, working on extracting frequent sequences, evolution patterns, and persistent/transient properties that correspond to this support, and then feeding this information as an input to the TLAT algorithm described in Section 5.7.

Finally, in order to define the hidden evolution patterns that the concepts exhibit in the time-stamped database, we assume that the temporal database is fairly complete at this

stage due to the lack of a temporal matching module; that is, we assume that there is no missing stage in the evolution patterns for all individual objects.

5.4 STEP Complexity Analysis

As shown in the design of the STEP algorithm, we make abundant use of index tables to store different database and sequence structures. In this monograph, index tables are implemented using Java's built-in *'Hashtable'* class. Hash tables are used to implement the *insert* and *find* operations in constant average time [Wei98]. On average, *Hashtable.put(Object key, Object value)*, and *Hashtable.get(Object key)* methods take two or three plots even when collisions tend to happen, which is a constant time. The worst time complexity of these operations differs however depending on the whether probing method used is linear, quadratic, or separate chain hashing. It is also important to pay attention to the load factor; otherwise, the constant time bounds are not meaningful [Wei98]. In our case, the load factor and other details of dealing with hash tables are handled automatically by Java, and that was not a major concern in the implementation.

Let n be the total number of rows or objects being observed in the temporal database, and let m represent the total number of attributes in the formal temporal context. The main complexity of the STEP algorithm lies in generating the frequent attribute sequences. Everything else such as populating and updating tables is, as explained above, constant in time. We will therefore focus on the complexity analysis of *generateF1()*, *generateF2()*, *generateF3()* and *generateF4()* modules.

The complexity of *generateF1()* results from traversing *propTable* index table, whose structure is shown in Table 5.2, that has m entries. For each attribute entry in *propTable*, we iterate through distinct objects having that attribute and increment the attribute support, *attSupport*. Let p represents the number of objects having a specific attribute. In the worst case when all objects have the same attribute, p equals n, and therefore the complexity of *generateF1()* module is in the order of $O(m * n)$. If we let $f1$ be the number

of frequent 1-sequence attributes, then obviously $f1 \leq m$, depending on the user-specified minimum support.

A call to *prepareDBTable()* module is performed in *generateF2()* to populate the two index tables *dbTable* and *jointDBTable*, whose structures are shown in Tables 5.3 and 5.4 respectively, by iterating over the *DB* vector. As mentioned before populating the two tables is constant, and therefore prepareDBTable has a linear $O(n)$ complexity. However, The main complexity of *generateF2()* results from performing a cross product of *F1*, the set of frequent sequences, with itself. A careful analysis of the *join(F1)* module leads to a complexity of $O(f1(f1-1)+f1*f1)$, which is in the order of $O(f1^2)$. In the worst case when $f1 = n$, module *join(F1)* has a complexity of $O(n^2)$. Let $s2 = s2e + s2t$ be the number of 2-sequence attributes, where $s2e = f1(f1-1)$ is the number of equality joined sequences that are stored in *eqlSeq*, and $s2t = f1^2$ is the number of temporally joined attributes that stored in *tplSeq*.

generateF2() finishes by calling *prepareEqlSeqTable()* and *prepareTplSeqTable()* modules to update index tables *eqlSeqTable* and *freqEqlSeqTable* with equality joined attributes, and *tplSeqTable* and *freqTplSeqTable* with temporally joined attributes. Equality joined attributes are separated by a space " ", to represent their occurrence at the same time, while temporally joined attributes are separated by an arrow "->" to represent their occurrences at different times.

Considering *prepareEqlSeqTable()* for analysis, it has a complexity analysis of $O(s2e*n*m)$ in the worst case. Note that it iterates over *dbTable* index table, whose structure is shown in Table 5.3, trying to look for the object that has the attribute sequence that is identical to the equality joined sequence in *eqlSeq*. In the worst case, an object could have all attributes present, and this is where m stems from. As stated before, since $s2e = f1(f1-1)$, then *prepareEqlSeqTable()* has a complexity of $O(f1^2*n*m)$, which is in the order of $O(m^3*n)$ in the worst case when $f1 = m$. A similar analysis to *prepareTplSeqTable()* leads a complexity of $O(s2t*n*m)$ in the worst case. Note that it

iterates over *jointDBTable* index table, whose structure is shown in Table 5.4 and which might have n entries in the worst case when every object is observed just one time, and looks for the object that has the attribute sequence identical to the temporally joined sequence in *tplSeq*. In the worst case, an object could have all attributes present or more in case that object happened to have persisting attributes, and this is where n and m stem from. As stated before, since $s2t = f1^2$, then *prepareTplSeqTable()* has a complexity of $O(f1^2 * n * m)$, which is also in the order of $O(m^3 * n)$ in the worst case when $f1 = m$.

Finally, *generateF2()* module has a total complexity of $O(n) + O(m^2) + 2 * O(m^3 * n)$ in the worst case, which is in the order of $O(m^3 * n)$. Let $f2 = f2e + f2t$ be the number of frequent 2-sequence attributes, where $f2e \leq s2e$ is the number of frequent equality joined sequences that are stored in *eqlSup*, and $f2t \leq s2t$ is the number of frequent temporally joined attributes that stored in *tplSup*. Note that in the worst case when all equality joined sequences are frequent then $f2e = s2e = f1(f1-1)$, and when all temporally joined attributes are frequent then $f2t = s2t = f1^2$.

Moving to *generateF3()*, a call to *prepareCaseA()* is performed to compute equality joined sequences, and calls to *prepareCaseB()*, *prepareCaseC()*, and *prepareCaseD()* are performed to compute temporally joined sequences. These four modules are nothing but a way of forming all possible permutations that generate all possible 3-sequence attributes. Both of *prepareCaseA()* and *prepareCaseD()* have a complexity of $O(f2e * f1)$. However, as stated mentioned earlier, since $f2e = f1(f1-1)$ in the worst case, then the complexity is $O(f1^3)$, which is in the order of $O(m^3)$ in the worst case when $f1 = m$. Similarly, Both of *prepareCaseB()* and *prepareCaseC()* have a complexity of $O(f2t * f1)$. However, as mentioned earlier, since $f2t = f1^2$ in the worst case, then the complexity is $O(f1^3)$, which is in the order of $O(m^3)$ in the worst case when $f1 = m$. Let $s3 = s3ea + s3tb + s3tc + s3td$ be the number of 3-sequence attributes, where $s3ea \approx s3td \approx f2e * f1$ is the number of equality joined sequences that are stored in

caseA and *caseD* respectively, and $s3tb \approx s3tc \approx f2t * f1$ is the number of temporally joined attributes that stored in *caseB* and *caseC* respectively.

A call in *generateF3()* to *updateSeqTables()* is then performed, which in turn calls *prepareEqlSeqTable()* on *caseA*, and *prepareTplSeqTable()* on *caseB*, *caseC*, and *caseD*. As discussed in *generateF2()*, *prepareEqlSeqTable()* will lead to a complexity of $O(s3ea * n * m)$ or $O(f2e * f1 * n * m)$ which is in the order of $O(m^4 * n)$ in the worst case when $f2e = f1(f1-1)$ and $f1 = m$. Similarly, *prepareTplSeqTable()* will lead to a complexity of $O(s3tb * n * m)$ and $O(s3tc * n * m)$ when called on *caseB* and *caseC* respectively, or $O(f2t * f1 * n * m)$ which is in the order of $O(m^4 * n)$ in the worst case when $f2t = f1^2$ and $f1 = m$. *prepareTplSeqTable()* will also lead to a complexity of $O(m^4 * n)$ in the worst case when called on *caseD*.

Let $f3 = f3ea + f3tb + f3tc + f3td$ be the number of frequent 3-sequence attributes, where $f3ea \le s3ea$ is the number of frequent equality joined sequences that are stored in *eqlSup*, and $f3tb \le s3tb$, $f3tc \le s3tc$, and $f3td \le s3td$ are the number of frequent temporally joined attributes that stored in *tplB*, *tplC*, and *tplD* respectively. Note that in the worst case when all equality joined sequences are frequent then $(f3ea = s3ea) \approx f2e * f1$, and when all temporally joined attributes are frequent then $(f3tb = s3tb) \approx f2t * f1$, $(f3tc = s3tc) \approx f2t * f1$, and $(f3td = s3td) \approx f2e * f1$.

At last *updateSeqTable()* calls *merge()* module that merges the contents of *tplB*, *tplC*, and *tplD*. In the worst case when non of these are empty, the complexity of *merge()* is $O(f3tb + f3tc + f3td)$ which is in the order of $O(m^3)$ in the worst case when $f2t = f1^2$ and $f1 = m$.

Finally, *generateF3()* module has a total complexity of $5 * O(m^3) + 4 * O(m^4 * n)$, which is in the order of $O(m^4 * n)$. In a similar fashion, we find that the complexity of *generateF4()* is in the order of $O(m^4 * n)$. Therefore, in general the complexity of the

STEP algorithm is polynomial and is of the order $O(m^{r+1} * n)$ for generating the frequent r-sequence attributes, where $r \leq m$. Note that when $r = m$, STEP's complexity becomes exponential. However, it is very unlikely that r will grow as big as m.

In cases where r is limited to 2 or 3, STEP's computational complexity is not a major concern. However, analyzing bigger data sets in which r tends to grow towards values of n indicates some implications of the computational complexity on the size of the problems that can be practically analyzed using the proposed algorithm. For example, assuming that every instruction takes one microsecond *(10^{-6} sec)* to execute, then it could take STEP 2.7 hours in the worst case to generate the frequent attribute sequences and extract the evolution patterns and the set of persistent/transient properties from a temporal database that has 10000 objects *($n = 10000$)*, 10 attributes *($m = 10$)*, and the maximal frequent attribute sequence consisting of 5 attributes only *($r = 5$)*.

The STEP algorithm has two advantages over the SPADE algorithm described in ([Zak97] and [Zak01]). The first is that STEP only makes one pass over the database at the beginning of the program while getting data from *patterns.dat*, and another pass during the call to *prepareEqlSeqTable()* module by iterating over the *dbTable* index table. The other advantage is that STEP does not store the template of every sequence in the table structure, which creates an additional overhead especially for sequences that are not frequent. It only computes and does not store the template of the sequences that are frequent when it is necessary while in modules *getPersistentProperties()* and *getOtherProperties()*. Implementation wise, the fact that all the modules described are reusable and can work with any number of attribute sequences of any length is also a third advantage.

On the other hand, SPADE outperforms STEP in terms of computation due to its scale up properties. In fact, extensive experiments show that the SPADE algorithm scales linearly with the average number of transactions per customer and the average number of items per transactions [Zak97]. This is due to the fact that SPADE uses equivalence classes to decompose the original problem of generating frequent sequences into smaller sub-

problems that can be solved independently and processed in main-memory [Zak01], while at the present STEP does not incorporate this feature. This is where STEP's polynomial complexity stems from.

5.5 Using Temporal Matching

Although the second phase of the STEP algorithm, described in Figure 5.1 and which consists of the design and implementation of a temporal matching module, is left as part of future work, we will briefly discuss in this section what this module does and how it can be useful. We also discuss the roles of the longest common subsequence, *LCS*, and the longest monotonic subsequence, *LMS*, in capturing common evolution patterns.

Temporal matching is the problem of matching observations to predefined temporal patterns or templates [TS01]. Temporal matching is needed in many applications such as model-based and medical diagnosis, plan recognition, computer vision, and temporal databases. Given a sequence of observations and some temporal evolution patterns, temporal matching consists of mapping observation times to particular points in the evolution patterns. In order to define the hidden evolution patterns that the concepts exhibit in the time-stamped database, we start with the patterns that have more stages, then we fit individual objects that do not show complete evolution stages to these patterns using temporal matching [TS01]. Note the presence of *'temporal matching'* in the *prediction* column of Table 2.5. Note also that temporal matching in this context differs from matching in temporal databases in that the latter aims at finding instances in the database that match a temporal query [DFS98].

We propose a dynamic programming approach for designing the temporal matching algorithm module. Assuming the evolution patterns in the time-stamped database are known, then the longest common subsequence (LCS) of all temporal attributes allows identifying the patterns of any given sequence. More specifically, the LCS determines how far an object is in an evolution, and is helpful in pattern matching since it permits to see how far we can go in finding the order of the attributes. Note that any attribute

sequence can exist in between or after the attributes of the known evolution patterns. Note also that this will only work provided we do not have missing stages in the evolution patterns for all individual observed objects.

Another way that helps in temporal matching is the use of the longest monotonic subsequence (LMS) of all the temporal attributes that is obtained by a simple mapping of the LCS algorithm. This mapping consists of sorting the elements of the temporal attribute sequence for which we want to find the LMS, and then passing the unsorted and the sorted list of the temporal attributes as the first and second arguments respectively to the LCS algorithm. Dealing with LMS is handy in the case we assign numbers to specific attributes in the attribute sequences. Using LCS and LMS reduces the number of patterns we are dealing with in the data set to *'common evolution patterns'*.

It is important to note that in temporal matching we might need to convert one string or attribute sequence to another by either inserting missing evolution stages in the pattern being scrutinized for temporal matching, or deleting adhoc patterns and evolution stages. This conversion process can be thought of the *'edit distance'* problem in dynamic programming which is defined as the cost of the least expensive transformation sequence that converts a string x to string y [CLR92]. In other words, the aim is to convert one attribute sequence to another using the minimum number of edits. Note also that changing one attribute sequence to another allows *'synonyms'* to be discovered among attribute sequences. For example, assuming that the characters *'c'*, *'j'*, *'a'*, and *'s'* stand for the attributes *child*, *juvenile*, *adult* and *senior* respectively then obviously changing the evolution pattern *'j a s'* to *'c a s'* is nothing but dealing with two synonym evolution patterns or attribute sequences.

The problem of temporal matching could have been framed as a constraint satisfaction problem [TS01] to verify the pattern formation stages. However, there is no guarantee that constraint satisfaction algorithms will outperform dynamic programming ones in terms of time complexity when it comes to dealing with large data sets due to the number of backtracking needed to satisfy the constraints.

We are usually interested in the patterns that occur frequently enough in the data set to be common. Assuming the characters *'d'*, *'e'*, *'j'*, *'r'*, *'s'*, and *'w'* stand for the attributes *drink*, *eat*, *jog*, *read*, *sleep* and *walk* respectively, then some of the possible cases of common patterns include:

- Two identical evolution patterns.
- Two evolution patterns that are reversed. An example is two evolution patterns having *'s j e d r'* and *'r d e j s'* as attribute sequences.
- Two evolution patterns where one is a subset of the other. An example is two evolution patterns having *'s j e d r'* and *'j e d'* as attribute sequences. Note that these patterns could include:
    - Evolution patterns that are part of a general trend. This general trend can be the LCS or LMS of all the objects' attributes. An example is having one evolution pattern whose attribute sequence *'e d'* is part of the longest monotonic attribute subsequence *'s j e d r w'*.
    - Evolution patterns that show missing evolution stages. An example is having one evolution pattern whose attribute sequence *'s j d w'* has fewer evolution stages than the ones found in the longest monotonic attribute sequence *'s j e d r w'*.

5.6 Generating Lattices

Algorithms for generating lattices usually consist of two parts: the first part constructs the lattice from a formal context, and the second draws the lattice diagram. For the first part, many simple algorithms have been proposed in the literature, the first of which were Wille's [Wil82], and Ganter's [Gan84] algorithms that are suitable for manual computation. As for the second part, the problem is not fully solved although there are nice solutions in practice, realized by the CERNATO software of NaviCon[4]. The easiest approach is that of an additive line diagram described in [GW99].

[4] 'Navigation and Concepts', NaviCon Decision suite, http://www.navicon.de

5.6.1 Algorithms for constructing a Lattice

Computing concept lattices is an important issue that has been recently investigated ([GR91], [MS89] and [YLCB96]). Several algorithms for constructing concept lattices from a binary relation have been described in ([Bor86], [Che69], [CR93], [Fay75], [Gan84], [GMA91], [GM94], [GMA95], [Mal62] and [Nor78]). Godin et al [GMA95] classify them into two categories: batch (non-incremental) algorithms ([Bor86], [Che69], [Fay75], [Gan84], and [Mal62]) and incremental ones ([CR93], [GM94], [GMA91], [GMA95] and [Nor78]). Incremental algorithms allow the visualization of the concept lattice as it is built by producing its corresponding Hasse diagram. This feature is important for the visualization of the lattice using computer-generated diagrams [Wil84].

The two algorithms presented in ([CR93] and [GMA91]) incrementally update the lattice and its corresponding Hasse diagram. Bordat's algorithm [Bor86] builds the Hasse diagram but is not incremental. Norris' algorithm [Nor78] is incremental but does not build the Hasse diagram. Godin et al [GMA95] present some algorithms that generate both the concept lattice and the Hasse diagram incrementally. Two incremental update algorithms and three batch algorithms from the related literature mentioned above are presented in [GMA95].

Many other recent algorithms have been described in ([Lin00], [NN97], [NR99], [STBPL00], [VM01] and [VML00]). Lindig [Lin00] presents LATTICE, an algorithm for concept analysis that computes concepts together with their explicit lattice structure, and compares it against Ganter's 'NEXTCONCEPT' algorithm and a third algorithm called 'CONCEPTS' that is used in Lindig's program *concepts* which gained some popularity in the past for computing concept lattices [Lin97]. Njiwoua et al [NN97] present ParGal, a polynomial time parallel algorithm to build a galois lattice that is based on Bordat's sequential algorithm [Bor86]. Nourine and Raynaud's algorithm [NR99] provides the best worst-case time complexity. Stumme et al [STBPL00] present TITANIC, a new algorithm for computing concept lattices that is

based on data mining techniques for computing frequent itemsets, and compare it with Ganter's 'Next Closure' algorithm [GR91]. Finally, Valtchev et al present an algorithm that is an improvement of Godin's [GMA95] algorithm in [VM01], and another one in [VML00] that shows a linear time complexity in the number of concepts, number of the attribute and the width of the lattice, and gives significantly better results than the other existing techniques in the case of a sparse context.

5.6.2 Algorithms for Drawing a lattice

Many algorithms have discussed the problem of drawing and handling concept lattices ([LWS86], [Wil89] and [Col00]). To design the TLAT algorithm - *Temporal LATtices*, we will rely on Stumme's algorithm [Stu00] for computing/drawing the concept lattice, and then change it to handle the addition of the directed temporal edges. Stumme's initial algorithm is described in Figure 5.27.

- From left to right, consider all intersections of each column extent with every column extent to the left of it. If the resulting extent is not already a column, add it as a column to the right end of the context. Repeat until the last (added) column is reached.
- Add a full column, unless there is already one. (Now each column stands for one concept)
- Draw a circle for the full column.
- Draw for each column, starting with the ones with the maximal number of crosses, a circle, and link it with a line to the circles where the column comprises the current column.
- Attach every attribute label to the circle of the corresponding column.
- Attach every object label to the circle laying exactly below the circles of the attributes in its intent.

Figure 5.27 How to compute/draw a concept lattice [Stu00]

5.7 TLAT: A New Algorithm for Drawing Temporal Lattices

The algorithm we present in this section is for generating temporal lattices with two kinds of edges described earlier in Chapter 4: temporal and non-temporal ones. We also aim in this algorithm to have the temporal edges drawn as high as possible in the hierarchy of concepts to make them generic enough to describe evolutions of particular classes rather than particular objects.

The different phases of The TLAT algorithm are detailed in Figure 5.28. Basically, TLAT takes the set of evolution patterns, persistent and transient properties extracted by the STEP algorithm described in the previous chapter, and shown in Table 5.11 as input, generates a temporal edge index table, *tplEdgeTable*, whose structure is shown in Table 5.12, as output and feeds that as an input to a graphics package that is concerned with drawing the concept lattice and adding to it the temporal edges that correspond to either a persistent property (a circular directed edge around the concept node) or an evolution pattern (a directed edge from one concept node to another). The TLAT algorithm neglects the transient properties, as they do not contribute to any significant trend that can be shown in the temporal concept lattice.

For constructing the temporal concept lattice from a temporal context in the graphics package, we will assume that it can be constructed using any of the algorithms mentioned in section 5.6.1. As for dealing with the drawing the temporal concept lattice, we will add to Stumme's algorithm [Stu00] to show the addition of the temporal directed edges. The design of the TLAT algorithm is shown in Figure 5.29. Note that for drawing circular directed temporal edges, we look for concept nodes whose attribute labels match the persistent attributes, while for drawing regular directed temporal edges, we have two scenarios. If the edge is assigned to more than one evolving object, we look for concept nodes whose attribute labels match the single attributes in the temporal edge in order to draw it as high as possible and make it more generic. However, if the edge is assigned to only one evolving object, then we look for concept nodes whose object labels match the evolutionary objects.

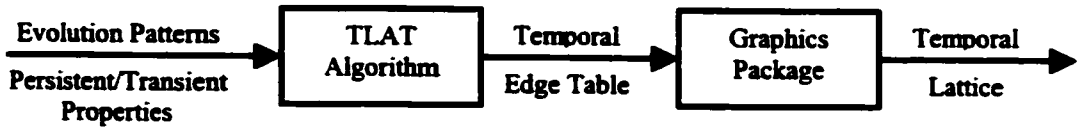

Figure 5.28 Different phases of TLAT algorithm

5.7.1 TLAT Design

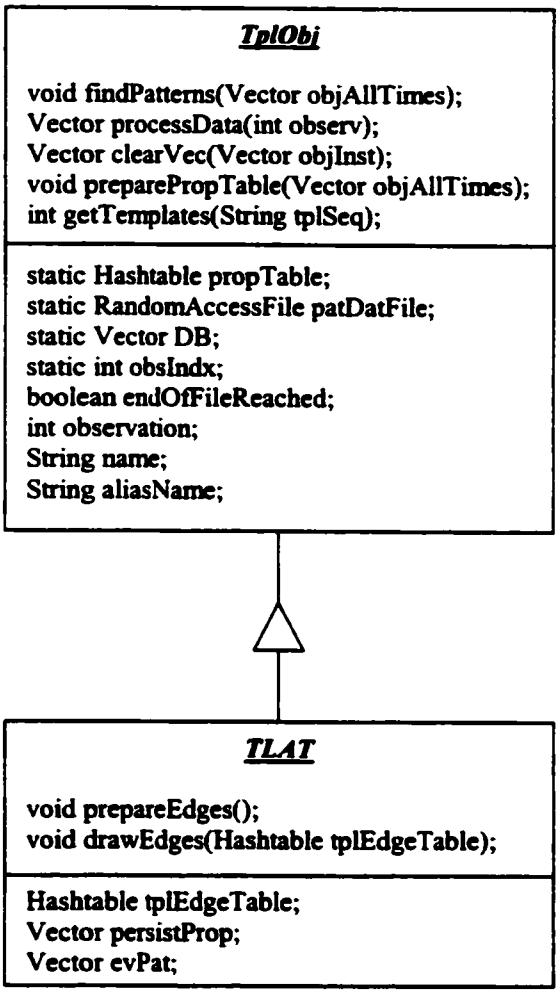

Figure 5.29 TLAT class hierarchy

5.7.2 TLAT Pseudo Code

We will describe in pseudo code the major two modules used in TLAT. These modules are *prerpareEdges()* and *drawEdges(Hashtable tplEdgeTable)*, in *TLAT.java* class. Note that the code for these modules can be found in *Appendix B* at the end of this monograph.

```
//-------------------------------------------------------
//- Prepare directed circular edges
//-------------------------------------------------------
for each persistent attribute att in persistProp
{
    objs = list of all objects having this persistent attribute;
    tplEdgeTable.put(att, objs);
}
//-------------------------------------------------------
//- Preparing directed edges from one node to another
//-------------------------------------------------------
for each evolution sequence evolSeq in evPat
{
    obj = object having evolSeq in evPat;
    for each single attribute att in evolSeq
    {
        objIDs = list of objects in propTable having this attribute;
        ID = time at which obj possesses att;
        aliasName = obj + ID;
        objSeq = objSeq + "->" + aliasName;
    }
    evolVec = all object sequences for evolSeq;
    tplEdgeTable.put(evolSeq, evolVec);
}
```

Figure 5.30 prepareEdges() module

```
for each edge edge in tplEdgeTable
{
    if edge is circular
    {
        //-------------------------------------------------------------
        //- Draw circular directed edges
        //-------------------------------------------------------------
        Look in the concept lattice for the node whose attribute label = edge;
        Draw a circular directed edge around that node;
    }
    else
    {
        //-------------------------------------------------------------
        //- Draw regular directed edges
        //-------------------------------------------------------------
        edgeObjs = (Vector)tplEdgeTable.get(edge);
        if ( edgeObjs.size() > 1)
        {
            //-----------------------------------------------------
            //- The edge is assigned to more than one evolving object
            //- Draw the edge for these objects based on attribute labels
            //- Draw the temporal edge as high as possible
            //-----------------------------------------------------
            previousAttLabel = "null";
            for each object label attLabel in edge
            {
                Look in the concept lattice for the node whose attribute label = attLabel;
                Draw a regular directed edge from the concept node
                whose attribute label = previousAttLabel to the current node
                whose attribute label = attLabel;
                previousAttLabel = attLabel;
            }
        }
        else
        {
            //-----------------------------------------------------
            //- The edge is assigned to only one evolving object
            //- Draw the edge for that object based on object labels
            //-----------------------------------------------------
            previousObjLabel = "null";
            for each object label objLabel in objLabels
            {
                Look in the concept lattice for the node whose object label = objLabel;
                Draw a regular directed egde from the concept node
                whose object label = previousObjLabel to the current node
                whose object label = objLabel;
                previousObjLabel = objLabel;
            }
        }
    }
}
```

Figure 5.31 drawEdges() module

Note also that Table 5.12 portrays the structure of the temporal edge index table, 'tplEdgeTable', according to our initial context studied in Table 4.3 with a minimum support set to 1.

male	[steve, adam]
dead	[mary]
female	[mary, nancy]
adult->senior	stevet1->stevet2
juvenile->adult	adamt1->adamt2
senior->dead	nancyt1->nancyt2

Table 5.12 'tplEdgeTable' table

5.8 TLAT Limitations

As mentioned previously, at the present the TLAT algorithm only takes the set of evolution patterns, persistent and transient properties extracted by the STEP algorithm described in the previous chapter, and shown in Table 5.11 as input, and then draws the concept lattice and adds to it the temporal edges. However, TLAT could be much enhanced to add more powerful features to the concept lattice in terms of visualization power if the second phase of the STEP algorithm shown in Figure 5.1 is implemented, and a temporal matching module is present to match individual objects that do not show complete evolution stages to the extracted patterns.

5.9 TLAT Complexity Analysis

For the part of algorithm that has to deal with constructing/drawing the concept lattice, the complexity was previously mentioned in Section 2.5. In what follows we will focus on analyzing the complexity of the two major modules of the TLAT algorithm,

prepareEdges() and *drawEdges(Hashtable tplEdgeTable)*, described in Figures 5.29 and 5.30 respectively.

Module *prepareEdges()* prepares two kinds of directed edges: circular ones drawn around a single node, and regular ones drawn from one node to another. To prepare circular edges, it has to iterate over the persistent properties vector, *persistProp*, while to prepare regular edges, it has to iterate over the evolution patterns vector, *evPat*. The STEP algorithm provides both *persistProp* and *evPat* as input to the TLAT algorithm. Let n be the total number of rows or objects being observed in the temporal database, and let m represent the total number of attributes in the formal temporal context. Let $p \leq m$ be the number of persistent properties populated in *persistProp*, where m is the total number of the attributes found in the temporal database. Note that in the worst case when all attributes are persistent, then $p = m$.

While iterating over *persistProp* vector to prepare circular directed edges, a temporal edge index table, *tplEdgeTable*, whose structure is shown in Table 5.12, is being populated by the persistent attribute as a key and a vector containing the objects having these attributes as a value for that key. Populating both of these structure requires a constant time as explained earlier in section 5.4. The complexity of this iteration is therefore $O(p)$.

The second iteration over the *evPat* vector to prepare regular directed edges takes the evolution pattern sequence(s) of every object having an evolution and tries to populate *tplEdgeTable* by the evolution temporal sequence as a key and a vector containing the object-ID sequences having this temporal evolution as a value for that key. Let e be the number of evolution pattern sequences found in evPat, and let a be the number of single attributes found in an evolution temporal sequence.

A look up of each of the single attributes is performed in *PropTable* index table whose structure is shown in Table 5.2, in order to find the ID of the object having this single attribute, which is the observation time, t_i, at which the object has been observed to

possess this attribute. In the worst case this lookup has a complexity of $O(a * m)$ when the set of the single attributes of an evolution temporal sequence form the set of all attributes in the formal context. Therefore, the complexity of this second iteration is in the order of $O(e * a * m)$. Since a is a constant with values that range between 2 and 10 maximum in a typical database, the complexity is in the order of $O(e * m)$.

Finally, the complexity of *prepareEdges()* is in the order of $O(p) + O(e * m)$, which is in the order of $O(e * m)$ in the worst case when $p = m$.

We are left with the *drawEdges()* module that is concerned with drawing the actual edges by iterating over *tplEdgeTable* index table, and getting the different edges of both kinds in this table. As mentioned earlier in Section 5.7, drawing circular directed temporal edges requires looking concept nodes whose attribute labels match the persistent attributes. Drawing regular directed temporal edges, however, has two scenarios. It requires looking for concept nodes whose attribute labels match the single attributes in the temporal edge if the edge is assigned to more than one evolving object, and looking for concept nodes whose object labels match the evolutionary objects if the edge is assigned to only one evolving object. Let $E = e + p$ be the number of edges found in *tplEdgeTable*.

Assuming that the attribute and object labels for the concept lattice nodes are stored in hash tables, then the cost of looking up a node that has a particular attribute or object label will be a constant c. In this case, the complexity of drawing each circular edge is $O(c * d)$, where d is also a constant representing the cost of physically drawing the arrow of the directed edge. Thus, this complexity is constant. As for the case of a regular edge, an iteration over the objects and their corresponding IDs is performed in order to specify the node whose object label matches the object and its ID if the edge is assigned to one evolving object only, while an iteration over the single attributes in the temporal edge is performed to draw the edge as high as possible in the concept hierarchy if it is assigned to more than one evolving object. Again, let a be the number of single attributes found in an

evolution temporal sequence. In both cases, the complexity of drawing regular edges is therefore $O(E * a)$, which is in the order of $O(e + p)$ since $E = e + p$ and a is a constant as mentioned earlier, or $O(e + m)$ in the worst case when $p = m$.

In conclusion, the complexity of the TLAT algorithm is $O(e * m) + O(e + m)$ which is of order $O(e * m)$.

5.10 Summary

In this chapter, we have discussed STEP and TLAT, two new algorithms for inferring sequential temporal properties and for drawing temporal lattices respectively. We have presented the design of the two algorithms, mentioned some of their limitations, and studied their time complexity. We also briefly described the tasks of a temporal matching module. It is important to note that since the STEP algorithm generates different trends for the same temporal context based on the variation of the support that the user specifies, the TLAT algorithm produces, based on the same support, different representations of the temporal lattices that differ in the direction of the temporal edges, both circular ones drawn around a given concept node and regular ones drawn from one concept node to another.

As a final remark, we discuss how the integration of FCA is done within the TLAT system. We assumed that a FCA tool inside the graphics package, shown in Figure 5.28, generates static lattices, and then these are integrated with the output of the TLAT module, which is the temporal edge table, *tplEdgeTable*, whose structure is shown in Table 5.12. Labeling the temporal edges/arcs that TLAT produces with the user-specified minimum support helps in the categorization of temporal edges in the temporal lattice based on this support. Changing the support from 1, and setting it to an entire class size or even to an arbitrary support level will result in different shapes of the temporal lattice. Having the graphics package drawing the temporal edges that are generated by TLAT through *tplEdgeTable* allows us therefore to trace the evolution patterns according to

their labeled support properly. At the end the temporal lattice produced by the graphics package, according to our initial temporal context shown in Table 4.3, will look something similar to Figure 5.32:

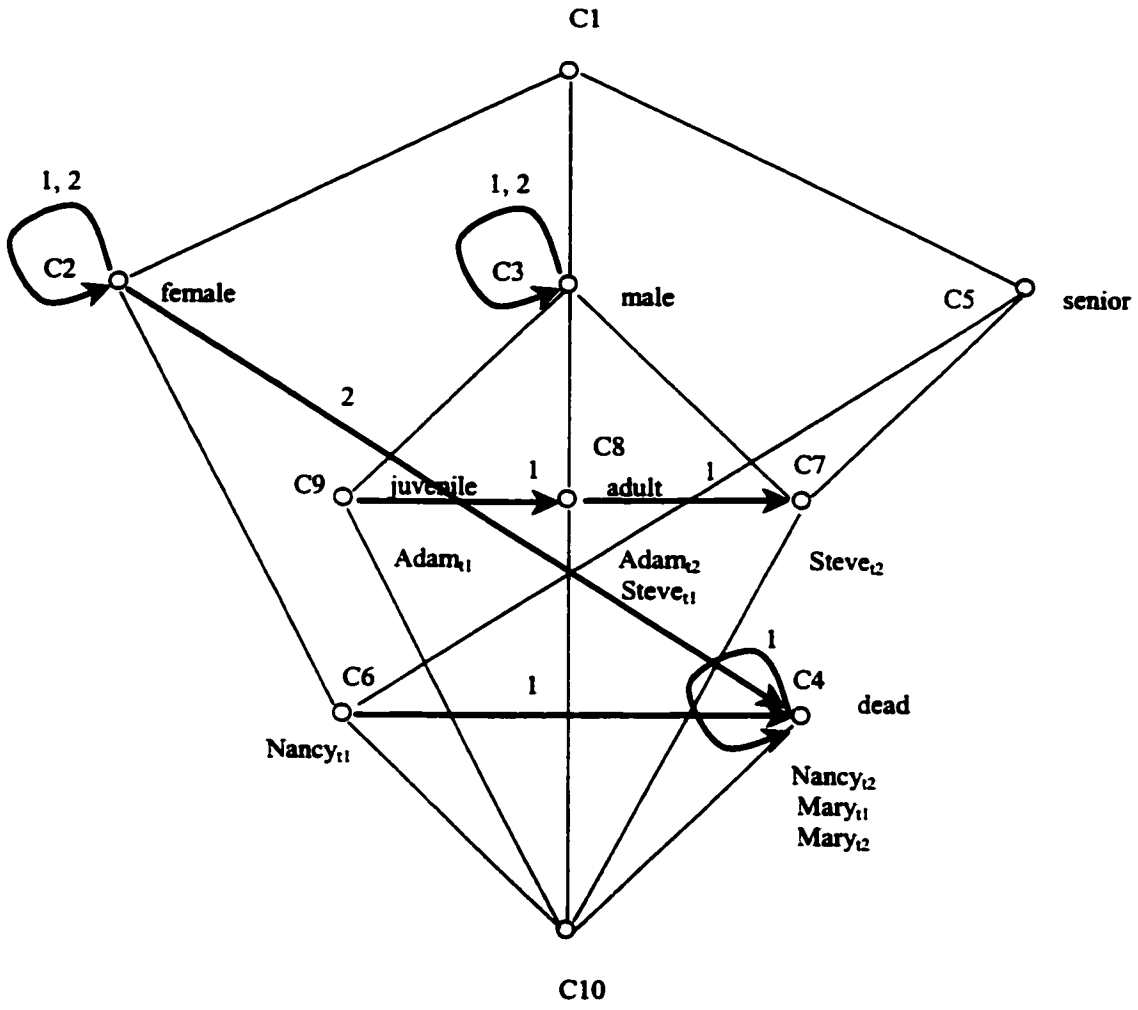

Figure 5.32 Final temporal lattice structure

Chapter 6
Applications and Future Work

6.1 Conclusions

This monograph has presented a useful temporal extension to FCA can be readily designed and implemented. The work that has been done includes:

- An algorithm for inferring temporal properties (STEP), and an algorithm for drawing temporal lattices (TLAT)
- A complexity analysis of these two algorithms
- An application of these algorithms to a time-stamped database

The extension presented is important since:

- Discovering useful temporal patterns rely on data visualization to complement data mining in the knowledge discovery process. Lattices, just like databases, are a good structure for containing these implicit temporal patterns.
- Visualization helps developing insights and deduces the hidden regularities in the data.
- Animation seems to provide proper visualization for temporal evolution. However such animations can be easily generated from the proposed lattices.
- Concept hierarchies provide a new tool for the study of the relationships between an interval and its subintervals.

This monograph also proposed a framework for discovering evolution patterns represented by concept lattices and analyzing temporal metamorphosis. The approach we used is a temporal extension of FCA. We would like to emphasize that the result is a more generic framework than just discovering trends from time-stamped orders or sequences. In fact, these orders could be, for example, the different sequential levels of

education or educational difficulty, the 'ph' level in a chemical reaction, the height from the ground for any given object, or a classification of time intervals/subintervals.

Answering temporal queries in temporal databases has been studied widely, but the scope of the work presented in this monograph is delimited by the presentation of a new approach for representing temporal evolutions and inferring temporal properties from time-stamped databases. It exploits some unique properties of patterns and trends that make the analysis and inferring process of these evolutions an efficient task. Our main intuition was that the classification of temporal properties we have proposed in Chapter 4 should reflect a mechanism for inferring them and consequently extracting evolution patterns.

In the rest of this chapter, we discuss potential TFCA applications as well as some limitations of the work presented in this monograph. Then, as a future research, we conclude by projecting on some interesting problems that remain open.

6.2 TFCA Applications

6.2.1 Phylogeny Applications

Explaining the evolutionary history of today's species and relating them in terms of common ancestors requires the construction of phylogenic trees, whose leaves represent these species and whose interior nodes represent hypothesized ancestors. One important aspect of phylogenic trees is the distance between ancestral objects to leaf objects, which can be interpreted as the 'time' it took for the ancestral object to evolve into the leaf object [SM97].

Input data for phylogeny construction can be classified into a *character state matrix*, an objects × characters matrix, in which each discrete character can have a finite number of states. The data relative to these characters can be placed in the matrix. An example of a character state matrix is shown in Table 6.1.

	Character				
Object	C1	C2	C3	C4	C5
A	1	1	0	0	0
B	0	0	1	0	1
C	1	1	0	0	1
D	0	1	1	1	0
E	1	1	0	0	1

Table 6.1 A character state matrix [SM97]

Formally speaking, a character state matrix is a matrix M with n rows (objects) and m columns (characters), where M_{ij} denotes the state that object i has for character j. The characters in Table 6.1 can be beak shape, number of fingers, presence or absence of a molecular restriction site, and breeding style.

One assumption of the character-based phylogeny reconstruction method is that characters can be inherited independently from one another. Another one is that all observed states for a given character should have evolved from one original state of the nearest common ancestor being studied. Such a character is called *homologous*. This information, if captured, can be used to draw temporal edges described in Section 4.5.1 among homologous characters in the corresponding phylogenic lattice.

6.2.2 Shoham's Proposition Types

Shoham ([SG88] and [Sho87]) represents temporal information in a logical formalism by associating 'proposition types' with time points and intervals. He claims to have constructed a 'richer and more flexible' categorization of proposition types than McDermott's fact/event dichotomy or Allen's property/event/process trichotomy. The way to distinguish among the different kinds of propositions is by specifying how the truth of a proposition over one interval is related to its truth over other intervals.

Shoham's interval ontological representation consists of six main proposition classes: the first is 'downward hereditary' where the proposition holds over all subintervals of a given interval whenever it holds over that interval itself. The second is 'upward hereditary' where the proposition holds over a nonpoint interval whenever it holds for all its proper subintervals. The third is 'liquid' where the proposition is both upward hereditary and downward hereditary. The fourth is 'concatenable' or 'clay-like' where the proposition holds over the union of two consecutive intervals whenever it holds over these two consecutive intervals. The fifth is 'gestalt' where the proposition never holds over two intervals, one of which properly containing the other. The sixth is 'solid' where the proposition never holds over two properly overlapping intervals.

One possible application is to use TFCA as a tool for the study of the relationship between an interval and its subintervals in order to have a classification of time intervals/subintervals, and to capture a precedence relationship between temporal points and intervals. The originality of this work resides in discovering taxonomies for time intervals.

6.3 Limitations

At present, STEP and TLAT algorithms do not classify intervals or express any visual insights about the relationship between an interval and its subintervals. While this task is deferred to a future work research, it is extremely essential to find ways that capture precedence relationships between temporal points and intervals, and have this temporal information displayed by temporal lattices.

In addition, these two algorithms do not handle conditional evolution patterns. All the evolution patterns that STEP extracts and feeds to TLAT are assumed to be unconditional ones (refer to Sections 4.6.1 and 4.6.2). One intuition toward dealing with this limitation is to label the temporal edges/arcs that TLAT produces with the user-specified minimum support. This way we will be able to categorize temporal edges in the temporal lattice by the minimum support that labels them. In the case where a group of objects manifest an

evolution pattern, then in the process of drawing the temporal edges at a higher level in the concept hierarchy, we might be able to capture the condition for evolution pattern, if it is conditional, by examining the attribute label of the concept node from which the temporal edges start.

6.4 Future Work

As a future work, different research directions can take place. One route is to investigate repetitive evolution patterns and study their effects on concept lattices and analysis of data. Examples of such repetitive patterns could be 'a person eats three times a day', 'the sun rises and sets one time a day', 'days and nights occur seven times a week'.

At the present time, there is no silver bullet to distinguish between evolution patterns that are recurring and those that are persisting. Working in this direction could lead to interesting temporal knowledge in studying the persistence/recurrence duration of events or evolution patterns. This is important because some temporal intervals might not be independent of one another, and might not be even sequential ([AH89], [All83] and [All84]). Under such circumstances, a classification of temporal intervals or of Shoham's propositions discussed in Section 6.2.2 comes in handy.

Another way is to accommodate the difference between evolution patterns, persistent and transient properties at both the individual level and the class level. A mechanism is also needed to differentiate between relative persistence and absolute persistence discussed in Section 4.6.4, and to decide whether relative or absolute persistence can start and/or end evolution stages in an evolution pattern.

Working on improving the STEP algorithm is also a fundamental direction to make it deal with equivalence classes ([Zak97] and [Zak01]) and benefit from partitioning the problem of enumerating all frequent sequences. Of course adding the temporal matching module, shown in Figure 5.1, as part of a second phase to the STEP algorithm can also

enhance the power of the TLAT algorithm to represent and express the flow of temporal knowledge.

Another track is to investigate a modal extension of FCA using the notion of temporal lattices. Such an extension would let us decide for example whether students who evolve from first year to second year standing should have 'necessarily', 'typically', or 'usually' taken a given course for that evolution to happen. A modal extension would have lots of applications in predictive data mining that searches for very strong patterns in big data that can generalize to accurate future decisions [WI98].

Still another related path is to examine some statistical techniques in TFCA. Such techniques could offer potential help in separating temporal trends, extracted from time-stamped databases, which follow an evolution pattern from those who do not. In order to capture "common or usual" evolution patterns of the kind discussed in Section 5.5, some statistical techniques have to be explored in order to define percentage thresholds that will differentiate between attribute sequences that support a specific evolution pattern, and sequences that do not follow that pattern.

References

[AH89] Allen, J. F., and Hayes, P. J. 1989. Moments and points in interval-based temporal logic. Computational Intelligence, vol. 5, no. 4, pp. 225-238.

[AIT93] Agrawal, R., Imielinski, T., and Swami, A. 1993. Mining association rules between sets of items in large databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data (May), Washington D. C., pp. 207-216.

[All83] Allen, J. F. 1983. Maintaining knowledge about temporal intervals. Communications of the ACM, (Nov.), vol. 26, no. 11, pp. 832-843.

[All84] Allen, J. F. 1984. Towards a general theory of action and time. Artificial Intelligence, vol. 23. Elsevier Science Publishers B. V., North-Holland, pp. 123-154.

[AS94] Agrawai, R., and Srikant, R. 1994. Fast algorithms for mining association rules. In Proceedings of 20th International Conference on Very Large Databases (Sept.), Santiago, Chile, pp. 478-499. Expanded Version Available as IBM Research Report RJ9839, June 1994.

[AS95] Agrawal, R., and Srikant, R. 1995. Mining sequential patterns. In Proceedings of the International Conference on Data Engineering (ICDE'95), Taipei, Taiwan (March). Expanded Version Available as IBM Research Report RJ9910, October 1994.

[BB98b] Bartel, H. -G., and Bruggemann, R. 1998. Application of formal concept analysis to structure-activity relationships. Fresenius Journal of Analytical Chemistry (5 Jan.), vol. 361, no. 1. Springer-Verlag, Berlin, Germany, pp. 23-28.

[Bir48] Birkhoff, G. 1948. Lattice theory. American Mathematical Society. Colloquium Publications, Vol. XXV, Second (Revised) Edition. Providence, Rhode Island, USA, 283

pp. Third Edition (1979), Library of Congress Catalog Card Number 66-23707, 418 pp. ISBN: 0821810251.

[Bis92] Bisson, G. 1992. Conceptual clustering in a first order logic representation. In Proceedings of the 10th European Conference on Artificial Intelligence, ECAI-92. John Wiley & Sons Ltd., Chichester, pp. 458-462.

[BN97] Bartel, H. -G., and Nofz, M. 1997. Exploration of NMR data of glasses by means of formal concept analysis. Chemometrics and Intelligent Laboratory Systems (Feb.), vol. 36, no. 1. Elsevier, Netherlands, pp. 53-63.

[Bor86] Bordat, J. P. 1986. Calcul pratique du treillis de galois d'une correspondance, Mathematiques et Sciences Humaines, vol. 96, pp. 31-47.

[BVS97] Bruggemann, R., Voigt, K., and Steinberg, C. E. W. 1997. Application of formal concept analysis to evaluate environmental databases. Chemosphere, vol. 35, no. 3. Elsevier Science Ltd., pp. 479-486.

[CE96] Cole, R. J., and Eklund, P. W. 1996. Text retrieval for medical discharge summaries using SNOMED and formal concept analysis. Australian Document Computing Symposium, pp 50-58.

[CE99a] Cole, R., and Eklund, P. 1999. Analyzing an email collection using formal concept analysis. Lecture Notes in Computer Science, 1999, Vol. 1704. Springer-Verlag, Berlin, Germany, pp. 309-315.

[CE99b] Cole, R., and Eklund, P. W. 1999. Scalability in formal concept analysis. Computational Intelligence (Feb.), vol. 15, no. 1. Blackwell Publishers, USA, pp. 11-27.

[CEG97] Cole, R., Eklund, P., and Groh, B. 1997. Dealing with large contexts in formal concept analysis: a case study using medical texts. In Proceedings of the Second

International Symposium on Knowledge Retrieval, Use, and Storage for Efficiency, KRUSE-97 (Aug.), Vancouver, B.C., Canada, pp. 151-164.

[CEW98a] Cole, R., Eklund, P., and Walker, D. 1998. Using conceptual scaling in formal concept analysis for knowledge and data discovery in medical texts. In Proceedings of the Second Pacific Asian Conference on Knowledge Discovery and Data Mining. World Scientific, pp. 378-379.

[CEW98b] Cole, R., Eklund, P., and Walker, D. 1998. Constructing conceptual scales in formal concept analysis. Research and Development in Knowledge Discovery and Data Mining. In Proceedings of the Second Pacific Asian Conference on Knowledge Discovery and Data Mining, PAKDD-98, (Melbourne, Vic., Australia, 15-17 Apr.). Lecture Notes in Computer Science, 1998, Vol. 1394. Springer-Verlag, Berlin, Germany, pp. 378-379.

[Cha94] Chalmers, D. J. 1994. The components of content. PNP Technical Report 94-04, Washington University. http://www.u.arizona.edu/~chalmers/papers/content.html.

[Che69] Chein, M. 1969. Algoritheme de recherche des sous-martices premieres d'une martice. Bull. Math. Soc. Sci. Math., R. S. Roumaine, vol. 13, pp. 21-25.

[CLR92] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1992. Introduction to algorithms. The MIT Electrical Engineering and Computer Science Series. MIT Press, McGraw-Hill Book Company. 1028 pp., ISBN: 0-07-013143-0.

[CM98] Chaudron, L., and Maille, N. 1998. 1st order logic formal concept analysis: from logic programming to theory. Linkoping Electronic Articles in Computer and Information Science, (23 Sept.), vol. 3, no. 13. Linkoping University Electronic Press, Linkoping, Sweden, pp. 1-14. ISSN: 1401-9841.

[Col00] Cole, R. 2000. Automatic layout of concept lattices using force directed placement and genetic algorithms. Australian Computer Science Conference (Jan). Available at: http://www.int.gu.edu.au/kvo/papers/gd.pdf.

[CR93] Carpineto, C., and Romano, G. 1993. Galois: an order-theoretic approach to conceptual clustering. In Proceedings of the Machine Learning Conference, pp. 33-40.

[DFS98] Dumas, M., Fauvet, M., and Scholl, P. 1998. Handling temporal grouping and pattern-matching queries in a temporal object model. In Proceedings of the Seventh Conference on Information and Knowledge Management, CIKM'98.

[DGM97] Das, G., Gunopulos, D., and Mannila, H. 1997. Finding similar time series. In Principles of Knowledge Discovery and Data Mining, PKDD'97.

[DLMRS98] Das, G., Lin, K., Mannila, H., Renganathan, G., and Smyth, P. 1998. Rule discovery from time series. In Proceedings of the Fourth International Conference on Knowledge Discovery in Data Mining, KDD'98. AAAI Press.

[Erd98] Erdmann, M. 1998. Formal concept analysis to learn from the sisyphus-III material. In Proceedings of the 11th Knowledge Acquisition for Knowledge-Based Systems, Workshop (KAW'98), (Banff, Canada). Available at: http://ksi.cpsc.ucalgary.ca /KAW/KAW98/erdmann.

[EW98] Eklund, P. W., and Wille, R. 1998. A multimodal approach to term extraction using a rhetorical structure theory tagger and formal concept analysis. In Proceedings of the 2nd International Conference on Multi-modal Communication, CMC/98, (Tilburg). pp. 171-175, ISBN: 90-9011386-X.

[Fay75] Fay, G. 1975. An algorithm for finite galois connections. Journal of Computational Linguistic and Languages, vol. 10, pp. 99-123.

[FC99] Frost, R. A., and Chitte, S. 1999. A new approach for providing natural-language speech access to large knowledge-bases. In Proceedings of the Pacific Association of Computational Linguistics Conference, PACLING '99. University of Waterloo, (Aug.), pp. 82-89.

[FCA00] FCA Bibliography. Publications of the Darmstadt Research Group on Formal Concept Analysis. Available at: http://www.mathematik.tu-darmstadt.de/ags/ag1/Literatur/literatur_en.html.

[Fel98] Fellbaum, C. 1998. WordNet: An Electronic Lexical Database. The MIT Press, Cambridge, MA. 423 pp., ISBN: 0-262-06197-X.

[FF98] Fernandez-Manjon, B., and Fernandez-Valmayor, A. 1998. Building educational tools based on formal concept analysis. Education and Information Technologies (Dec.), vol. 3, no. 3-4. Kluwer Academic Publishers, Netherlands, pp. 187-201.

[Fis87a] Fisher, D. H. 1987. Knowledge acquisition via incremental conceptual clustering. Machine Learning, Vol. 2, pp. 139-172.

[Fis87b] Fisher, D. H. 1987. Conceptual clustering, learning from examples, and inference. In Proceedings of the 4th International Workshop on Machine Learning, Vol. 2, No. 2. Morgan-Kaufman Publishers Inc., Los Altos, California, pp. 38-49.

[Fis87c] Fisher, D. H. 1987. Improving inference through conceptual clustering. In Proceedings of the AAAI Conference, (July). Seattle, Washington, pp. 461-465.

[Fre92] Frege, G. 1892. "*Uber Sinn und Bedeutung*," Zeitschrift fur Philosophie und philosophische Kritik, vol. 100, pp. 25-50. Translated as "On sense and reference", in Translations from the Philosophical Writings of Gottlob Frege (P.T. Geach & M. Black, eds.), Basil Blackwell, Oxford, 1977, pp. 56-85.

[Gan84] Ganter, B. 1984. Two basic algorithms in concept analysis. FB4-Preprint 831 (June), TH Darmstadt, Darmstadt, Germany.

[Gan99] Ganter, B. 1999. Attribute exploration with background knowledge. ORDAL '96, (Ottawa, Ont., Canada, 5-9 Aug. 1996). Theoretical Computer Science (6 April), vol. 217, iss. 2. Elsevier Science Ltd., Netherlands, pp. 215-233.

[Gen89] Gennari, J. H. 1989. Focused concept formation. In Proceedings of the 5th International Workshop on Machine Learning. Morgan Kaufman Publishers, San Mateo, CA, pp. 379-382.

[GK99] Ganter, B., and Krausse, R. 1999. Pseudo models and prepositional Horn inference. FB4-Preprint, TH Darmstadt, Darmstadt, Germany.

[GLF89] Gennari, J. H., Langley, P., and Fisher, D. H. 1989. Models of incremental concept formation. Artificial Intelligence, Vol. 40, No. 1-3, pp. 11-61.

[GM94] Godin, R., and Missaoui, R. 1994. An incremental concept formation approach for learning from databases. Theoretical Computer Science (Oct.), Special Issue on Formal Methods in Databases and Software Engineering, vol. 133, no. 2, pp. 387-419.

[GMA91] Godin, R., Missaoui, R., and Alaoui, H. 1991. Learning algorithms using a galois lattice structure. In Proceedings of the Third International Conference on Tools for Artificial Intelligence, San Jose, CA. IEEE Computer Society Press, pp. 22-29.

[GMA95] Godin, R., Missaoui, R., and Alaoui, H. 1995. Incremental concept formation algorithms based on galois (concept) lattices. Computational Intelligence, vol. 11, no. 2, pp. 246-267.

[God89] Godin, R. 1989. complexite' de structures de treillis. Ann. Sc. Math., Vol. 13, No. 1. Quebec, Montreal, Canada.

[Goo65] Goodman, N. 1965. Fact, Fiction and Forecast. Oxford: Oxford University Press. Indianapolis: Bobbs-Merrill.

[GR91] Ganter, B., and Reuter, K. 1991. Finding all closed sets: a general approach. Order, vol. 8. Kluwer Academic Publishers, The Netherlands, pp. 283-290.

[Gro95] Groh, B. 1995. Distributive concept exploration for Windows. FB4-Preprint, TH Darmstadt, Darmstadt, Germany.

[Gus97] Gusfield, D. 1997. Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, USA. 534 pp., ISBN: 0-521-58519-8.

[GW99] Ganter, B., and Wille, R. 1999. Formal concept analysis: mathematical foundations. Springer-Verlag, Berlin, Heidelberg, New York. 284 pp. ISBN: 3-540-62771-5.

[Hay96] Hayes, P. J. 1996. A catalog of temporal theories. Technical Report, UIUC-BI-AI-96-01, The Beckman Institute, University of Illinois.

[HCC93] Han, J., Cai, Y., and Cercone, N. 1993. Data-driven discovery of quantitative rules in relational databases. IEEE Transactions on Knowledge and Data Engineering, vol. 5, no. 1, pp. 29-40.

[HSWW00] Hereth, J., Stumme, G., Wille, R., and Wille, U. 2000. Conceptual knowledge discovery and data analysis. FB4-Preprint 2092 (May), TU Darmstadt, Darmstadt, Germany. In Proceedings of the 8th International Conference on Conceptual Structures, ICCS'00, (Darmstadt, Germany, 14-18 Aug.). Conceptual Structures: Logical, Linguistic, and Computational Structures. Lecture Notes in Artificial Intelligence, Vol. 1867. Springer-Verlag, Berlin, Germany, pp. 418-434. ISBN: 3-540-67859-X.

[Hul91] Hultsch, E. 1991. Structure of informations on medical trials. Classification, Data Analysis, and Knowledge Organization. Models and Methods with Applications. In Proceedings of the 14th Annual Conference of the Gesellschaft fur Klassifikation, (Marburg, Germany, 12-14 March 1990). Springer-Verlag, Berlin, Germany, pp. 277-283.

[KS94] Krone, M., and Snelting, G. 1994. On the inference of configuration structures from source code. In Proceedings of the 16th International Conference on Software Engineering, ICSE-16, (Sorrento, Italy, 16-21 May). IEEE Comput. Soc. Press, Los Alamitos, CA, USA, pp. 49-57.

[Leb86] Lebowitz, M. 1986. Concept learning in a rich input domain: generalization-based memory. Machine Learning: An Artificial Intelligence Approach, Vol. II. Morgan-Kaufman Publishers Inc., Los Altos, California, pp. 193-214.

[Leb87] Lebowitz, M. 1987. Experiments with incremental concept formation: UNIMEM. In Proceedings of the 4th International Workshop on Machine Learning, Vol. 2, No. 2. Morgan-Kaufman Publishers Inc., Los Altos, California, pp. 103-308.

[Lin97] Lindig, C. 1997. Concepts. http://www.eecs.harvard.edu/~lindig/software/download/concepts-0.3e.tar.gz. Open Source Implementation of Concept Analysis in C.

[Lin99] Lindig. C. 1999. Algorithmen zur begriffsanalyse und ihre anwendung bei softwarebibliotheken. PhD Dissertation, D-38106 (Nov.), Technical University of Braunschweig, Braunschweig, Germany. (In German).

[Lin00] Lindig, C. 2000. Fast concept analysis. In Proceedings of the 8th International Conference on Conceptual Structures, ICCS'00, (Darmstadt, Germany, 14-18 Aug.). Working with Conceptual Structures - Contributions to ICCS 2000. Shaker-Verlag, Aachen, Germany, pp. 152-161. ISBN: 3-8265-7669-1.

[LS97] Lindig, C., and Snelting, G. 1997. Assessing modular structure of legacy code based on mathematical concept analysis. In Proceedings of the 1997 International Conference on Software Engineering, ICSE 97, (Boston, MA, USA, 17-23 May). ACM, New York, NY, USA, pp. 349-359.

[LSW86] P. Luksch, M. Skosrksy, and R. Wille. 1986. On drawing concept lattices with a computer. In W. Gaul and M. Schader, editors, Classifcation as a tool of research. North Holland, Amsterdam, pp. 269-274.

[Lu97] Lu, Y. 1997. Concept hierarchy in data mining: specification, generation and implementation. Master of Science Thesis (Dec.), School of Computing Science. Simon Fraser University, Burnaby, BC, Canada.

[LW88] Luksch, P., and Wille, R. 1988. Formal concept analysis of paired comparisons. Classification and Related Methods of Data Analysis. In Proceedings of the 1st Conf. on the Intl. Federation of Classification Societies, IFCS, (Aachen, West Germany, 29 June-1 July 1987). North-Holland, Amsterdam, Netherlands, pp. 567-575.

[LW95] Lehmann, F., and Wille, R. 1995. A triadic approach to formal concept analysis. Conceptual Structrues: Applications, Implementation and Theory. In Proceedings of the 3rd Intl. Conf. on Conceptual Structures, ICCS'95, (Santa Cruz, CA, USA, 14-18 Aug). Lecture Notes in Computer Science, 1995, Vol. 954. Springer-Verlag, Berlin, Germany, pp. 32-43.

[Mal62] Malgrange, Y. 1962. recherche des sous-martices premieres d'une martice a coefficients binaires applications a certains probleme de graphs. In Proceedings of the Deuxieme Congres de l'AFCALTI, Gauthier-Villars, pp. 231-242.

[Man97] Mannila, H. 1997. Methods and problems in data mining. In Proceedings of the International Conference on Database Theory, (Greece, Jan.). Springer-Verlag, pp. 41-55.

[McD82] McDermott, D. 1982. A temporal logic for reasoning about processes and plans. Cognitive Science, vol. 6, no. 2, pp. 101-155.

[Mil90] Miller, G. A. 1990. WordNet: An On-Line Lexical Database. International Journal of Lexicography, Volume 3, Number 4.

[MKB86] Mott, J. L., Kandel, A., and Baker, T. P. 1986. Discrete mathematics for computer scientists and mathematicians. Second Edition. A Reston Book. Prentice-Hall, Englewood Cliffs, NJ. 751 pp., ISBN: 0-8359-1391-0.

[MR97] Mannila, H., and Ronkainen, P. 1997. Similarity of event sequences. In Proceedings of the Fourth International Workshop on Temporal Representation and Reasoning (Time), (May 10-11). IEEE Computer Society Press. Daytona Beach, Florida, USA. ISBN: 0-8186-7937-9.

[MS83a] Michalski, R. S., and Stepp, R. E. 1983. Learning from observation: conceptual clustering. Machine Learning: An Artificial Intelligence Approach, Vol. 1, Chapter 11, Tioga Publishing Company, pp. 331-363.

[MS83b] Michalski, R. S., and Stepp, R. E. 1983. Automated construction of calssification: conceptual clustering versus numerical taxonomy. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 5, pp. 396-410.

[MS89] Missikoff, M., and Scholl, M. 1989. An algorithm for insertion into a lattice: application to type classification. In Proceedings of 3rd International Conference FODO'89. Lecture Notes in Computer Science, Vol. 367. Springer-Verlah, Heidelberg, pp. 64-82.

[MTV94] Mannila, H., Toivonen, H., and Verkamo, A. I. 1994. Efficient algorithms for discovering association rules. In Proceedings of the AAAI Workshop on Knowledge Discovery in Databases, KDD'94, (July). AAAI Press, pp. 181-192.

[MTV97] Mannila, H., Toivonen, H., and Verkamo, A. I. 1997. Discovery of frequent episodes in event sequences. Series of Publications C, Report C-1997-15, (Feb.). University of Helsinki, Department of Computer Science, Finland, 45 pp.

[NK93] Nowinski, G., and Krebs, V. 1993. A knowledge acquisition and processing strategy based on formal concept analysis. In Proceedings of the IFAC Symposium on Artificial Intelligence in Real-Time Control, 1992, (Delft, Netherlands, 16-18 June 1992). Selected Papers from the IFAC/IFIP/IMACS Symposium. Pergamon, Oxford, UK, pp. 103-108.

[NN97] Njiwoua, P., and Nguifo, E. M. 1997. A parallel algorithm to build concept lattice. In Proceedings of the Fourth Groningen International Information Technology Conference for Students, pp. 103-107.

[Nor78] Norris, E. M. 1978. An algorithm for computing the maximal rectangles in binary relation. Revue Roumaine de Mathematiques Pures et Appliquees, vol. 23, no. 2. Bucharest, pp. 243-250.

[NR99] Nourine, L., and Raynaud, O. 1999. A fast algorithm for building lattices. Information Processing Letters, vol. 71, no. 5-6, pp. 199-204. Technical Report, Laboratoire d'Informatique, de Robotique, de Micro-Electronique de Montpellier, LIRMM, France, 1998.

[NTF01] Neouchi, R., Tawfik, A. Y., and Frost, R. A. 2001. Towards a temporal extension of formal concept analysis. In Proceedings of the 14th Canadian Conference on Artificial Intelligence (June 7-9), AI'2001, Ottawa, Canada. Lecture Notes in Artificial Intelligence, no. 2056. Springer-Verlag, Berlin.

[Pei31] Peirce, C. 1931. Collected papers of Charles Sanders Peirce. Harvard University Press, Cambridge, MA.

[Ped93] Pedersen, G. S. 1993. A browser for bibliographic information retrieval, based on an application of lattice theory. In Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, (Pittsburgh, PA, USA, June). ACM, pp. 270-279.

[Pre97] Prediger, S. 1997. Logical scaling in formal concept analysis. Conceptual Structures: Fulfilling Peirce's Dream. In Proceedings of the 5th International Conference on Conceptual Structures, ICCS'97, (Seattle, WA, USA, 3-8 Aug.). Lecture Notes in Computer Science, 1997, Vol. 1257. Springer-Verlag, Berlin, Germany, pp. 332-341.

[Ron98] Ronkainen, P. 1998. Attribute similarity and event sequence similarity in data mining. Licentiate Thesis, Series of Publications C, Report C-1998-42, (Oct.). University of Helsinki, Department of Computer Science, Finland, 98 pp.

[PBTL98] Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. 1998. Pruning closed itemset lattices for association rules. In Proceedings of the BDA French Conference on Advanced Databases, (Oct.).

[PBTL99] Pasquier, N., Bastide, Y., Taouil, R., and Lakhal, L. 1999. Efficient mining of association rules using closed itemset lattices. Journal of Information Systems, vol. 24, no. 1, pp. 25-46.

[Pri98] Priss, U. 1998. Relational concept analysis: semantic structures in dictionaries and lexical databases. D 17 (Dissertation TU Darmstadt 1996). Doctoral Thesis. Shaker Verlag, Aachen, 116 pp. ISBN: 3-8265-4099-9.

[Pri01] Priss, U. 2001. "Formal Concept Analysis Homepage", http://php.indiana.edu/~upriss/fca/fca.html.

[Ram98] Ramakrishnan, R. 1998. Database Management Systems. WCB McGraw-Hill. 741 pp. ISBN: 0-07-050775-9.

[SA96] Srikant, R., and Agrawal, R. 1996. Mining sequential patterns: generalizations and performance improvements. In Proceedings of the Fifth International Conference on Extending Database Technology (EDBT'96), Avignon, France (Nov.). Expanded Version Available as IBM Research Report RJ9994, December 1995.

[Sar99] Sarbo, J. J. 1999. Formal conceptual structure in language. In Proceedings of Computing Anticipatory Systems, CASYS'98. Woodbury, New York, pp. 289-300. AIP Conference Proceedings 465.

[SC99] Schmitt, I., and Conrad, S. 1999. Restructuring object-oriented database schemata using formal concept analysis. Informatik Forschung und Entwicklung, vol. 14, no. 4. Springer-Verlag, Germany, pp. 218-226.

[SCTC96] Shadbolt, N., Crow, L., Tennison, J., and Cupit, J. 1996. Sisyphus III resource page. Available at: http://www.psyc.nott.ac.uk/aigr/research/ka/SisIII/

[SG88] Shoham, Y., and Goyal, N. 1988. Temporal reasoning in artificial intelligence. In Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence (Dec.). Morgan Kaufmann Publishers, pp. 693. ISBN: 0-934613-69-9.

[She99] Shen, Y. 1999. Formal concept analysis and its application in software engineering/reuse. 60-510 background reading survey report, (Nov.). University of Windsor, Windsor, Ontario, Canada, pp. 1-86.

[She00] Shen, Y. 2000. Concept-based retrieval of object-oriented class components for software reuse. M.Sc. Thesis, Computer Science Department, University of Windsor, Windsor, Ontario, Canada. 115 pp.

[Sho85] Shoham, Y. 1985. Ten requirements for a theory of change. New Generation Computing, vol. 3. OHMSHA. LTD., and Springer-Verlag, pp. 467-477.

[Sho87] Shoham, Y. 1987. Temporal logics in AI: semantical and ontological considerations. Artificial Intelligence, Vol. 33, pp. 89-104. Elsevier Science Publishers B.V., North-Holland.

[Sho88] Shoham, Y. 1988. Reasoning about change: time and causation from the standpoint of artificial intelligence. The MIT Press, Cambridge, Massachusetts, London, England. 201 pp., ISBN: 0-262-19269-1. PhD Thesis, Yale University, Computer Science Department, 1986.

[SM86] Stepp, R. E., and Michalski, R. S. 1986. Conceptual clustering: inventing goal-oriented classification of structured objects. Machine Learning: An Artificial Intelligence Approach, Vol. II. Morgan-Kaufman Publishers Inc., Los Altos, California, pp. 471-498.

[SM97] Setubal, J., and Meidanis, J. 1997. Introduction to computational molecular biology. PWS Publishing Company, MA, USA. 296 pp., ISBN: 0-534-95262-3.

[Sne98] Snelting, G. 1998. Concept analysis - a new framework for program understanding. ACM SIGPLAN Notices (July), (Montreal, Que., Canada, 16 June), vol. 33, no. 7. ACM, USA, pp. 1-10.

[Sne00] Snelting, G. 2000. Software reengineering based on concept lattices. In Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (Zurich, Switzerland, 29 Feb.-3 March). IEEE Comput. Soc, Los Alamitos, CA, USA, pp. 3-10.

[SR97] Siff, M., and Reps, T. 1997. Identifying modules via concept analysis. In Proceedings of the IEEE International Conference on Software Maintenance, ICSM'97, (Bari, Italy, 1-3 Oct.). IEEE Computer Society Press, Washington, DC, pp. 170-179. Revised Version Available as Technical Report TR-1337. Computer Sciences Department, University of Wisconsin, Madison, WI, August 1998.

[ST98] Snelting, G., and Tip, F. 1998. Reengineering class hierarchies using concept analysis. In Proceedings of the Sixth International Symposium on the Foundations of Software Engineering, ACM-SIGSOFT'98, (Lake Buena Vista, FL, 3-5 Nov.). Software Engineering Notes, vol. 23, no. 6. ACM, USA, pp. 99-110.

[STBPL00] Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., and Lakhal, L. 2000. Fast computation of concept lattices using data mining techniques. In Proceedings of the 7th International Workshop on Knowledge Representation meets Databases, KRDB'00, (21 August). Berlin, Germany.

[Stu95] Stumme, G. 1995. Knowledge acquisition by distributive concept exploration. Supplementary proceedings of the third international conference on conceptual structures: Applications, Implementation and Theory, ICCS'95, (Santa Cruz, CA, USA, 14-18 August). Springer-Verlag, Berlin, Germany, pp. 98-111.

[Stu96] Stumme, G. 1996. Exploration tools in formal concept analysis. Ordinal and symbolic data analysis. Studies in classification, data analysis, and knowledge organization 8. Springer, Heidelberg, pp. 31-44.

[Stu97] Stumme, G. 1997. Concept exploration - a tool for creating and exploring conceptual hierarchies. Conceptual Structures: Fulfilling Peirce's Dream. LNAI no. 1257, Springer, Berlin, pp. 318-331.

[Stu98a] Stumme, G. 1998. On-line analytical processing with conceptual information systems. In Proceedings of the 5th International Conference on Foundations of Data Organization (Nov.), vol. 12-13. Kluwer Academic Publishers, Netherlands, pp. 117-126.

[Stu98b] Stumme, G. 1998. Distributive concept exploration-a knowledge acquisition tool in formal concept analysis. KI-98: Advances in Artificial Intelligence. In Proceedings of the 22nd Annual German Conference on Artificial Intelligence (Bremen, Germany, 15-17 Sept.). Lecture Notes in Computer Science, 1998, Vol. 1504. Springer-Verlag, Berlin, Germany, pp. 117-128.

[Stu99a] Stumme, G. 1999. Acquiring expert knowledge for the design of conceptual information systems. In Proceedings of the 11th European Workshop on knowledge Acquisition, Modeling, and Management, (Dagstuhl, 26-29 May), (submitted). Lecture Notes in Artificial Intelligence, no. 1621. Springer-Verlag, Berlin-Heidelberg, pp. 271-286.

[Stu99b] Stumme, G. 1999. Conceptual knowledge discovery with frequent concept lattices. FB4-Preprint 2043 (June), TU Darmstadt, Darmstadt, Germany.

[Stu00] Stumme, G. 2000. Knowledge discovery. Course Transparencies. Universitaet Karlsruhe. http://www.aifb.uni-karlsruhe.de/Lehrangebot/Winter2000-01/kdd00_01/.

[SWW98] Stumme, G., Wille, R., and Wille, U. 1998. Conceptual knowledge discovery in databases using formal concept analysis methods. Principles of Data Mining and Knowledge Discovery. In Proceedings of the Second European Symposium, PKDD'98, (Nantes, France, 23-26 Sept.). Lecture Notes in Computer Science, 1998, Vol. 1510. Springer-Verlag, Berlin, Germany, pp. 450-458.

[Taw97] Tawfik, A. Y. 1997. Changing times: An investigation in probabilistic temporal reasoning. PhD Dissertation, University of Saskatchewan, Saskatoon, Saskatchewan, Canada. 145 pp.

[TBPSL00] Taouil, R., Bastide, Y., Pasquier, N., Stumme, G., and Lakhal L. 2000. Mining bases for association rules based on formal concept analysis. In Proceeding of the 14th European Conference on Artificial Intelligence, ECAI'00, (20-25 August). Berlin, Humboldt University, Germany.

[TN99] Tawfik, A. Y., and Neufeld, E. 1999. Changing times: a casual theory of probabilistic temporal reasoning. J. EXPT. THEOR. ARTIF. INTELL. Vol. 11, pp. 3-21.

[Tru94] Trudel, A. 1994. A temporal structure that distinguishes between the past, present, and future. In Proceedings of the Time-94 International Workshop on Temporal Representation and Reasoning (May 4). Florida AI Research Symposium, FLAIRS-94, pp. 146-152. Pensacola, FL. 175 pp., ISBN: 0-7731-0278-7.

[TS01] Tawfik, A. Y., and Scott, G. 2001. Temporal matching under uncertainty. In Proceedings of the Eighth International Workshop on Artificial Intelligence and Statistics.

[VM01] Valtchev, P., Missaoui, R. 2001. Building concept (galois) lattices from parts: generalizing the incremental approach. In Proceedings of the 9th International Conference on Conceptual Structures, ICCS'01, (July 30 - Aug. 3). Stanford University, California, USA. Lecture Notes in Artificial Intelligence, Vol. 2120. Springer-Verlag, Berlin, Germany, pp. 290-303.

[VML00] Valtchev, P., Missaoui, R., and Lebrun P. 2000. A fast algorithm for building the hasse diagram of a galois lattice. Colloque LACIM 2000, Combinatoire, Informatique et Applications, (7-10 Sept.). Laboratoire de Combinatoire et d'Informatique Mathematique. Universite du Quebec a Montreal, Montreal, Canada, pp. 293-306.

[VW95] Vogt, F., and Wille, R. 1995. TOSCANA-a graphical tool for analyzing and exploring data. Knowledge Organization, vol. 22, no. 2. Germany, pp. 78-81. Graph

Drawing. In Proceedings of the DIMACS International Workshop, GD'94, (Princeton, NJ, USA, 10-12 Oct. 1994). Lecture Notes in Computer Science, 1994, Vol. 894. Springer-Verlag, Berlin, Germany, pp. 226-233. ISBN: 3-540-58950-3.

[Wei98] Weiss, M. A. 1998. Data structures and algorithm analysis in Java. Addison-Wesley Longman, Inc., (Oct.), 780 pp. ISBN: 0-201-54991-3.

[WI98] Weiss, S. M., and Indurkhya, N. 1998. Predictive data mining: a practical guide. Morgan Kaufmann Publishers, Inc. San Francisco, California, 228 pp., ISBN: 1-55860-403-0.

[Wil82] Wille, R. 1982. Restructuring lattice theory: an approach based on hierarchies of concepts. Ordered Sets. D. Reidel Publishing Company, Dordrecht-Boston, pp. 445-470.

[Wil84] Wille, R. 1984. Line diagrams of hierarchical concept systems. In International Classification, vol. 11, no. 2, pp. 77-86.

[Wil89] Wille, R. 1989. Lattices in data analysis: How to draw them with a computer. Algorithms and Order. Kluwer Academic Publishers, The Netherlands, pp. 33-58.

[Wil92] Wille, R. 1992. Concept lattices and conceptual knowledge systems. Computers & Mathematics with Applications (March-May), vol. 23, no. 6-9. Pergamon Press plc, UK, pp. 493-515.

[Wil95] Wille, R. 1995. The basic theorem of triadic concept analysis. Order - Journal of the Theory of Ordered Sets and Its Applications (31 Jan), vol. 12, no. 2. Kluwer Academic Publishers, Netherlands, pp. 149-158.

[WL97] Waiyamai, K., and Lakhal, L. 1997. Towards an object database approach for managing concept lattices. In Proceedings of the 16th International Conference on

Conceptual Modeling, ER'97, (Los Angeles, CA, USA, 3-5 Nov.). Lecture Notes in Computer Science, 1997, Vol. 1331. Springer-Verlag, Berlin, Germany, pp. 299-312.

[WNRR99] Willett, D., Neukirchen, C., Rottland, J., and Rigoll, G. 1999. Refining tree-based state clustering by means of formal concept analysis, balanced decision trees and automatically generated model-sets. In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP99, (Phoenix, AZ, USA, 15-19 March), vol. 2. IEEE, Piscataway, NJ, USA, pp. 565-568.

[YLCB96] Yahia, A., Lakhal, L., Cicchetti, R., and Bordat, J. P. 1996. iO2: an algorithmic method for building inheritance graphs in object database design. In Proceedings of the 15th International Conference on Conceptual Modeling, ER'96, (Cottbus, Germany, 7-10 Oct.). Lecture Notes in Computer Science, 1996, Vol. 1157. Springer-Verlag, Berlin, Germany, pp. 422-437.

[Zak97] Zaki, M. J. 1997. Fast mining of sequential patterns in very large databases. Technical Report 668 (Nov.), The University of Rochester, Computer Science Department, Rochester, New York.

[Zak01] Zaki, M. J. 2001. SPADE: an efficient algorithm for mining frequent sequences. In Machine Learning Journal, Special Issue on Unsupervised Learning (Jan./Feb.), Vol. 42, Nos. 1/2. Kluwer Academic Publishers, Boston, The Netherlands, pp. 31-60.

[Zic91] Zickwolff, M. 1991. Rule exploration: first order logic in formal concept analysis. PhD Thesis, TH Darmstadt University, Darmstadt, Germany. Short English Version, Preprint nr. 1580, June 1993.

Appendix A

STEP Algorithm Code

```
import java.util.*;
import java.io.*;

//-------------------------------------------------------------
//-  (C) Copyright 2001 Rabih A. Neouchi
//-------------------------------------------------------------

//-------------------------------------------------------------
//-  STEP - Sequential TEmporal Properties
//-------------------------------------------------------------
public class TplObj extends Object
{
    static Hashtable propTable = new Hashtable();
    static RandomAccessFile  patDATFile;
    static Vector DB = new Vector();
    boolean  endOfFileReached = false;
    static int obsIndx = 0;

    int observation;
    String name;
    String aliasName;
    String oneLine;

    TplObj() {}
    TplObj(String objName, int observ)
    {
        name = objName;
        observation = observ;
    }

    public void findPatterns(Vector objAllTimes) {return;}

    public Vector processData(int observ)
    {
        Vector objAllTimes = new Vector();
        objAllTimes.removeAllElements();
        //-------------------------------------------------------
        // Open patterns data files (*.DAT files)
        //-------------------------------------------------------
        try { patDATFile = new RandomAccessFile( "patterns.dat","r"); }
        catch(IOException ioe)
        {
            System.out.println("\nerror: in openning the <patterns.dat> file...");
            System.exit(0);
        }
        //-------------------------------------------------------
        //- position read at offset
        //-------------------------------------------------------
```

```
        try { patDATFile.seek( 0 ); }
        catch(IOException ioe) { }

        for (int i = 0; i<obsIndx; i++)
        {
            //-----------------------------------------------------
            //- Seeking to the right 'offset' position
            //-----------------------------------------------------
            try { oneLine = patDATFile.readLine(); }
            catch(IOException ioe) { }
        }
        for (int i = obsIndx; i<(obsIndx+observ); i++)
        {
            //-----------------------------------------------------
            //- Starting at 'offset' read a line of data
            //-----------------------------------------------------
            try { oneLine = patDATFile.readLine(); }
            catch(IOException ioe) { }

            oneLine = oneLine.substring(0, oneLine.length()-1);
            //-----------------------------------------------------
            //- Make a StringTokenzier of data line and process each field
            //-----------------------------------------------------
            StringTokenizer tokenizer = new StringTokenizer( oneLine, " " );
            aliasName = ( tokenizer.nextToken() );
            //-----------------------------------------------------
            //- Prepare vector data
            //-----------------------------------------------------
            Vector objInst = new Vector();
            objInst.addElement(aliasName);
            while (tokenizer.hasMoreTokens())
                objInst.addElement(tokenizer.nextToken());
            objAllTimes.addElement(objInst);
            //-----------------------------------------------------
            //- Prepare DB vector data
            //-----------------------------------------------------
            Vector dbObj = new Vector();
            Vector atts = (Vector)objInst.clone();
            atts.removeElementAt(0);
            Vector clratts = clearVec(atts);
            dbObj.addElement(aliasName);
            dbObj.addElement(clratts);
            DB.addElement(dbObj);
        }
        obsIndx = obsIndx + observ;
        return objAllTimes;
    }

    public Vector clearVec(Vector objInst)
    {
        Vector result = new Vector();

        for (int i = 0; i<objInst.size(); i++)
        {
            if ( ((objInst.elementAt(i)).toString()).compareTo( "null" ) == 0 )
```

```
            {
                objInst.removeElementAt(i);
                i--;
            }
        }
        result = (Vector)objInst.clone();
        return result;
    }

    public void preparePropTable(Vector objAllTimes)
    {
        for (int i=0; i<objAllTimes.size(); i++)
        {
            Vector objInst = (Vector)objAllTimes.elementAt(i);
            //-----------------------------------------------------
            //- Clear from "null" entries
            //-----------------------------------------------------
            Vector clrObjInst = clearVec(objInst);
            //-----------------------------------------------------
            //- Prepare 'propTable' hash table
            //-----------------------------------------------------
            for (int j=1; j<clrObjInst.size(); j++)
            {
                String att = (String)clrObjInst.elementAt(j);
                if ( (! propTable.containsKey(att)) )
                    propTable.put(att, new Vector());

                String aliasName = (String)objInst.elementAt(0);
                String obj = aliasName.substring(0, aliasName.length()-2);
                String ID = aliasName.substring(aliasName.length()-2, aliasName.length());

                Vector objID = (Vector)propTable.get(att);
                objID.addElement(obj);
                objID.addElement(ID);
                propTable.put(att, objID);
            }
        }
    }

    public int getTemplates(String tplSeq)
    {
        //-----------------------------------------------------
        //- look only for A->B, A->B->C
        //- A, B, C should satisfy evolution condition
        //- ignore A->A, A->AB and the like
        //-----------------------------------------------------

        Vector atts = new Vector();

        int idx = tplSeq.indexOf(" ");
        if ( idx != -1 ) return 0;
        idx = tplSeq.indexOf("->");
        if ( idx != -1 )
        {
            boolean distinct = true;
```

```
                StringTokenizer tok = new StringTokenizer(tplSeq, "->");
                while ( tok.hasMoreTokens() && distinct )
                {
                    String att = (String)tok.nextToken();
                    if ( ! atts.contains(att) )
                        atts.addElement(att);
                    else
                        distinct = false;
                }
                if ( ! distinct ) return 0;
                return 1;
            }
            return 0;
        }
    }
```

```
import java.util.*;

//-------------------------------------------------------------
//-  (C) Copyright 2001 Rabih A. Neouchi
//-------------------------------------------------------------
public class Patterns extends TplObj
{
    static Hashtable dbTable = new Hashtable();
    Hashtable jointDBTable = new Hashtable();
    Hashtable eqlSeqTable = new Hashtable();
    Hashtable freqEqlSeqTable = new Hashtable();
    Hashtable tplSeqTable = new Hashtable();
    Hashtable freqTplSeqTable = new Hashtable();
    Hashtable evolveTable = new Hashtable();
    Vector persistProp = new Vector();
    Vector transientProp = new Vector();
    Vector evPat = new Vector();

    int attSupport;
    int minSupport;
    int eqlSeqSupport;
    int tplSeqSupport;
    String dumName = "";

    Patterns() {}
    Patterns(int support)
    {
        minSupport = support;
    }

    public boolean distinct(String obj)
    {
        if ( obj.compareTo(dumName)!= 0 )
        {
            dumName = obj;
            return true;
        }

        return false;
    }

    public Vector generateF1(Hashtable propTable)
    {
        Vector result = new Vector();

        Enumeration atts = propTable.keys();
        //-------------------------------------------------------
        //- each element in atts is an attribute string
        //-------------------------------------------------------
        while (atts.hasMoreElements())
        {
            attSupport = 0;
            String att = (String)atts.nextElement();
            Vector objIDs = (Vector)propTable.get(att);
```

```
            //-----------------------------------------------------
            //- Iterating through distinct objs
            //-----------------------------------------------------
            for (int i=0; i<objIDs.size(); i+=2)
            {
                String obj = (String)objIDs.elementAt(i);

                if ( distinct(obj) )
                    attSupport++;
            }
            if (attSupport >= minSupport)
                result.addElement(att);
        }
        dumName = "";

        return result;
    }

    public void prepareDBTable(Vector DB)
    {
        for (int i=0; i<DB.size(); i++)
        {
            //-----------------------------------------------------
            //- Preparing 'db' hash table
            //-----------------------------------------------------
            Vector dbObj = (Vector)DB.elementAt(i);
            String objName = (String)dbObj.elementAt(0);
            Vector objAtts = (Vector)dbObj.elementAt(1);
            dbTable.put(objName, objAtts);
            //-----------------------------------------------------
            //- Preparing 'jointDB' hash table
            //-----------------------------------------------------
            String aliasName = objName.substring(0, objName.length()-2);
            if ( distinct(aliasName) )
                jointDBTable.put(aliasName, new Vector());
            //-----------------------------------------------------
            //- tplAtts is a vector of vectors containing attributes
            //-----------------------------------------------------
            Vector tplAtts = (Vector)jointDBTable.get(aliasName);
            tplAtts.addElement(objAtts);
            jointDBTable.put(aliasName, tplAtts);
        }
        dumName = "";
    }

    public Vector join(Vector F1)
    {
        Vector eqljoin = new Vector();
        Vector tplJoin = new Vector();
        Vector result = new Vector();

        for (int i=0; i<F1.size(); i++)
        {
            String att = (String)F1.elementAt(i);
```

```
            Vector temp1 = (Vector)F1.clone();
            Vector temp2 = (Vector)F1.clone();
            temp1.removeElementAt(i);
            for (int j=0; j<temp1.size(); j++)
            {
                //-------------------------------------------------
                //- No duplicates guaranteed (AB)
                //-------------------------------------------------
                String seqStr = att + " " + (String)temp1.elementAt(j);
                eqljoin.addElement(seqStr);
            }
            for (int k=0; k<temp2.size(); k++)
            {

                //-------------------------------------------------
                //- No duplicates guaranteed (A->B, A->A)
                //-------------------------------------------------
                String tplSeqStr = att + "->" + (String)temp2.elementAt(k);
                tplJoin.addElement(tplSeqStr);
            }
        }

        result.addElement(eqljoin);
        result.addElement(tplJoin);
        return result;
    }

    public Vector prepareEqlSeqTable(Vector eqlSeq)
    {
        //-------------------------------------------------
        //- prepares 'eqlSeq' hash table (NOT FREQUENT)
        //- prepares 'freqEqlSeq' hash table (FREQUENT)
        //- returns vector containing (FREQUENT) eqlSeq, support
        //-------------------------------------------------

        Vector result = new Vector();

        for (int i=0; i<eqlSeq.size(); i++)
        {
            String eqlSeqInst = (String)eqlSeq.elementAt(i);
            //-------------------------------------------------
            //- do not go for unwanted iterations
            //-------------------------------------------------
            if ( ! eqlSeqTable.containsKey(eqlSeqInst) )
            {
                eqlSeqSupport = 0;
                Vector mObjs = new Vector();
                Vector attVec = new Vector();
                StringTokenizer eqlTokenizer = new StringTokenizer( eqlSeqInst, " " );
                while (eqlTokenizer.hasMoreTokens())
                    attVec.addElement(eqlTokenizer.nextToken());
                eqlSeqTable.put(eqlSeqInst, new Vector());
                boolean found = false;

                Enumeration objs = dbTable.keys();
```

```
//-------------------------------------------------------
//- each element in objs is an object string
//-------------------------------------------------------
while (objs.hasMoreElements())
{
    String obj = (String)objs.nextElement();
    String alias = obj.substring(0, obj.length()-2);
    Vector atts = (Vector)dbTable.get(obj);
    Vector idxVec = new Vector();
    for (int j=0; j<attVec.size(); j++)
    {
        String att = (String)attVec.elementAt(j);
        if ( atts.contains(att) )
            idxVec.addElement(new Integer(atts.indexOf(att)));
    }
    //-------------------------------------------------------
    //- all attributes are found
    //-------------------------------------------------------
    if (idxVec.size() == attVec.size())
    {
        boolean desOrder = false;
        int prvIdx = -1;
        Enumeration attIdx = idxVec.elements();
        while ( (attIdx.hasMoreElements()) && (!desOrder) )
        {
            int curIdx = ((Integer)attIdx.nextElement()).intValue();
            if (curIdx < prvIdx)
                desOrder = true;
            prvIdx = curIdx;
        }
        //-------------------------------------------------------
        //- all indices are in ascending order
        //-------------------------------------------------------
        if (!desOrder)
        {
            //-------------------------------------------------------
            //- in case an the eqlSeq for an object is found
            //- in the intension of that object at one time,
            //- check intensions of that object at other times
            //- to populate 'eqlSeqTable' properly, but
            //- do not increment support if it is found again
            //-------------------------------------------------------
            if ( distinct(alias) )
                eqlSeqSupport++;
            //-------------------------------------------------------
            //- Preparing 'eqlSeq' hash table
            //-------------------------------------------------------
            Vector objList = (Vector) eqlSeqTable.get(eqlSeqInst);
            objList.addElement(obj);
            mObjs = (Vector)objList.clone();
            eqlSeqTable.put(eqlSeqInst, objList);
        }
    }
}
dumName = "";
```

```
                //-----------------------------------------------------
                //- preparing 'freqEqlSeq' hash table (FREQUENT)
                //-----------------------------------------------------
                if (eqlSeqSupport >= minSupport)
                {
                    freqEqlSeqTable.put(eqlSeqInst, mObjs);
                    result.addElement(eqlSeqInst);
                    result.addElement(new Integer(eqlSeqSupport));
                }

            }
        }

        return result;
    }

    public Vector prepareTplSeqTable(Vector tplSeq)
    {
        //-----------------------------------------------------------
        //- prepares 'tplSeq' hash table (NOT FREQUENT)
        //- prepares 'freqTplSeq' hash table (FREQUENT)
        //- returns vector containing (FREQUENT) tplSeq, support
        //-----------------------------------------------------------

        Vector result = new Vector();

        for (int i=0; i<tplSeq.size(); i++)
        {
            String tplSeqInst = (String)tplSeq.elementAt(i);
            //-----------------------------------------------------
            //- do not go for unwanted iterations
            //-----------------------------------------------------
            if ( ! tplSeqTable.containsKey(tplSeqInst) )
            {
                tplSeqSupport = 0;
                Vector objs = new Vector();
                Vector attVec = new Vector();
                StringTokenizer tplTokenizer = new StringTokenizer( tplSeqInst, "->" );
                while (tplTokenizer.hasMoreTokens())
                    attVec.addElement(tplTokenizer.nextToken());
                tplSeqTable.put(tplSeqInst, new Vector());
                Enumeration sObjs = jointDBTable.keys();
                while ( sObjs.hasMoreElements() )
                {
                    String sObj = (String)sObjs.nextElement();
                    Vector mAtts = (Vector)jointDBTable.get(sObj);
                    Enumeration attEn = attVec.elements();
                    //-----------------------------------------------------
                    //- Enumeration atts contains vectors of attributes
                    //-----------------------------------------------------
                    Enumeration atts = mAtts.elements();
                    boolean found = false;
                    while ( attEn.hasMoreElements() )
                    {
```

```
String att = (String)attEn.nextElement();
while ( (atts.hasMoreElements()) && (!found) )
{
    Vector oneTime = (Vector)atts.nextElement();
    //-------------------------------------------------------
    //- attribute is of the form AB or A->B
    //-------------------------------------------------------
    if ( (att.indexOf(" ") != -1) || (att.indexOf("->") != -1) )
    {
        if ( att.indexOf(" ") != -1 )
        {
            //-------------------------------------------------------
            //- equality join
            //-------------------------------------------------------
            int idx = att.indexOf(" ");

            Vector temp1 = new Vector();
            for (int j=0; j<oneTime.size(); j++)
            {
                String tAtt = (String)oneTime.elementAt(j);
                Vector temp2 = (Vector)oneTime.clone();

                for (int k=0; k<=j; k++)
                    temp2.removeElementAt(0);
                for (int l=0; l<temp2.size(); l++)
                {
                    String seqStr = tAtt + " " + (String)temp2.elementAt(l);
                    temp1.addElement(seqStr);
                }
            }
            oneTime = (Vector)temp1.clone();
        }
        else if ( att.indexOf("->") != -1 )
        {
            //-------------------------------------------------------
            //- temporal join
            //-------------------------------------------------------
            int idx = att.indexOf("->");
            StringTokenizer inAtt = new StringTokenizer(att,
                                String.valueOf(att.charAt(idx)));

            int inVec = attVec.indexOf(att);
            attVec.removeElementAt(inVec);
            while ( inAtt.hasMoreTokens() )
            {
                att = inAtt.nextToken();
                attVec.insertElementAt(att, inVec);
                inVec++;
            }
            attEn = attVec.elements();
        }
    }

    if ( oneTime.contains(att) )
    {
```

```
                        if ( ! attEn.hasMoreElements() )
                            found = true;
                        else
                            att = (String)attEn.nextElement();
                    }
                }
                if ( !found ) break;
            }
            if ( found )
            {
                tplSeqSupport++;
                Vector objList = (Vector)tplSeqTable.get(tplSeqInst);
                objList.addElement(sObj);
                objs = (Vector)objList.clone();
                tplSeqTable.put(tplSeqInst, objList);
            }
        }
        //-------------------------------------------------
        //- preparing 'freqTplSeq' hash table (FREQUENT)
        //-------------------------------------------------
        if (tplSeqSupport >= minSupport)
        {
            freqTplSeqTable.put(tplSeqInst, objs);
            result.addElement(tplSeqInst);
            result.addElement(new Integer(tplSeqSupport));
        }
    }
}

    return result;
}

public Vector generateF2(Vector F1)
{
    Vector result = new Vector();

    prepareDBTable(DB);
    //System.out.println(jointDBTable.toString());
    Vector seqList = join(F1);
    Vector eqlSeq = (Vector)seqList.elementAt(0);
    Vector tplSeq = (Vector)seqList.elementAt(1);
    Vector eqlSup = prepareEqlSeqTable(eqlSeq);
    Vector tplSup = prepareTplSeqTable(tplSeq);

    result.addElement(eqlSup);
    result.addElement(tplSup);
    return result;
}

public Vector prepareCaseA(Vector F1, Vector eqlAtt)
{
    Vector result = new Vector();

    //-------------------------------------------------
    //- preparing case A
```

```
        //-------------------------------------------------------
        for (int i=0; i<eqlAtt.size(); i+=2)
        {
            String lStr = "";
            String seq1 = (String)eqlAtt.elementAt(i);
            StringTokenizer tok = new StringTokenizer( seq1, " " );
            while (tok.hasMoreTokens())
                lStr = tok.nextToken();
            for (int j=0; j<F1.size(); j++)
            {
                String seq2 = (String)F1.elementAt(j);
                if ( lStr.compareTo(seq2)!=0 )
                {
                    String seq = seq1 + " " + seq2;
                    result.addElement(seq);
                }
            }
        }

        return result;
    }

    public Vector prepareCaseB(Vector F1, Vector tplAtt)
    {
        Vector result = new Vector();

        //-------------------------------------------------------
        //- preparing case B
        //-------------------------------------------------------
        for (int i=0; i<tplAtt.size(); i+=2)
        {
            String lStr = "";
            String seq1 = (String)tplAtt.elementAt(i);
            StringTokenizer tok = new StringTokenizer( seq1, "->" );
            while (tok.hasMoreTokens())
                lStr = tok.nextToken();
            for (int j=0; j<F1.size(); j++)
            {
                String seq2 = (String)F1.elementAt(j);
                if ( lStr.compareTo(seq2)!=0 )
                {
                    String seq = seq1 + " " + seq2;
                    result.addElement(seq);
                }
            }
        }

        return result;
    }

    public Vector prepareCaseC(Vector F1, Vector tplAtt)
    {
        Vector result = new Vector();

        //-------------------------------------------------------
```

```
        //- preparing case C
        //-----------------------------------------------------
        for (int i=0; i<tplAtt.size(); i+=2)
        {
            String seq1 = (String)tplAtt.elementAt(i);
            for (int j=0; j<F1.size(); j++)
            {
                String seq2 = (String)F1.elementAt(j);
                String seq = seq1 + "->" + seq2;
                result.addElement(seq);
            }
        }

        return result;
    }

    public Vector prepareCaseD(Vector F1, Vector eqlAtt)
    {
        Vector result = new Vector();

        //-----------------------------------------------------
        //- preparing case D
        //-----------------------------------------------------
        for (int i=0; i<eqlAtt.size(); i+=2)
        {
            String seq1 = (String)eqlAtt.elementAt(i);
            for (int j=0; j<F1.size(); j++)
            {
                String seq2 = (String)F1.elementAt(j);
                String seq = seq1 + "->" + seq2;
                result.addElement(seq);
            }
        }

        return result;
    }

    public Vector merge(Vector tplB, Vector tplC, Vector tplD)
    {
        Vector result = new Vector();

        if ( !tplB.isEmpty() )
        {
            for (int i=0; i<tplB.size(); i++)
                result.addElement(tplB.elementAt(i));
        }
        if ( !tplC.isEmpty() )
        {
            for (int i=0; i<tplC.size(); i++)
                result.addElement(tplC.elementAt(i));
        }
        if ( !tplD.isEmpty() )
        {
            for (int i=0; i<tplD.size(); i++)
                result.addElement(tplD.elementAt(i));
```

```
        }

        return result;
    }

    public Vector updateSeqTables(Vector caseA, Vector caseB, Vector caseC, Vector caseD)
    {
        //-------------------------------------------------------
        //- updates eqlSeqTable and tplSeqTable
        //- updates freqEqlSeqTable and freqTplSeqTable
        //-------------------------------------------------------
        Vector result = new Vector();
        Vector tplSup = new Vector();

        Vector eqlSup = prepareEqlSeqTable(caseA);
        Vector tplB = prepareTplSeqTable(caseB);
        Vector tplC = prepareTplSeqTable(caseC);
        Vector tplD = prepareTplSeqTable(caseD);
        tplSup = merge(tplB, tplC, tplD);

        result.addElement(eqlSup);
        result.addElement(tplSup);
        return result;
    }

    public Vector generateF3(Vector F2, Vector F1)
    {
        //-------------------------------------------------------
        //- Consider 4 cases: A|BC, A|B->C, A->(B->C), A->BC AND
        //- Reciprocal cases: AB|C, A->B|C, (A->B)->C, AB->C
        //- Be careful of duplicates
        //-------------------------------------------------------
        Vector caseA = new Vector();
        Vector caseB = new Vector();
        Vector caseC = new Vector();
        Vector caseD = new Vector();
        Vector result = new Vector();

        Vector eqlAtt = (Vector)F2.elementAt(0);
        Vector tplAtt = (Vector)F2.elementAt(1);

        if ( !eqlAtt.isEmpty() )
        {
            caseA = prepareCaseA(F1, eqlAtt);
            caseD = prepareCaseD(F1, eqlAtt);
        }
        if ( !tplAtt.isEmpty() )
        {
            caseB = prepareCaseB(F1, tplAtt);
            caseC = prepareCaseC(F1, tplAtt);
        }

        result = updateSeqTables(caseA, caseB, caseC, caseD);

        return result;
```

```
}

public Vector generateF4(Vector F3, Vector F1)
{
    Vector result = new Vector();

    result = generateF3(F3, F1);

    return result;
}

public int getTemplate(String tplSeq)
{
    //-------------------------------------------------------
    //- look only for A->A, A->A->A
    //- ignore A->B, A->AB and the like
    //-------------------------------------------------------
    int idx = tplSeq.indexOf(" ");
    if ( idx != -1 ) return 0;
    idx = tplSeq.indexOf("->");
    if ( idx != -1 )
    {
        boolean same = true;
        StringTokenizer tok = new StringTokenizer(tplSeq, "->");
        String compare = tok.nextToken();
        while ( tok.hasMoreTokens() && same )
        {
            String att = (String)tok.nextToken();
            if ( att.compareTo(compare) != 0 )
                same = false;
        }
        if ( ! same ) return 0;
        return 1;
    }

    return 0;
}

public Vector getPersistentProperties(Hashtable freqTplSeqTable)
{
    Vector result = new Vector();

    Enumeration freqTpl = freqTplSeqTable.keys();
    while ( freqTpl.hasMoreElements() )
    {
        String tplSeq = (String)freqTpl.nextElement();
        int pat = getTemplate(tplSeq);
        if (pat == 1)
        {
            int idx = tplSeq.indexOf("->");
            String att = tplSeq.substring(0, idx);
            Vector objs = (Vector)freqTplSeqTable.get(tplSeq);
            result.addElement(att);
            result.addElement(objs);
            persistProp.addElement(att);
```

```
            }
        }

        return result;
    }

    public void organizeEvolveTable(Hashtable evolveTable)
    {
        //-------------------------------------------------------------
        //- organizing 'evolve' hash table 'values'
        //- to make transitions easier to detect
        //-------------------------------------------------------------
        Enumeration evObjs = evolveTable.keys();
        while ( evObjs.hasMoreElements() )
        {
            String obj = (String)evObjs.nextElement();
            Vector evolVec = (Vector)evolveTable.get(obj);
            Vector temp2 = (Vector)evolVec.clone();
            Enumeration evol = evolVec.elements();
            while ( evol.hasMoreElements() )
            {
                String evSeq = (String)evol.nextElement();
                Vector temp1 = (Vector)evolVec.clone();
                temp1.removeElement(evSeq);
                Enumeration rest = temp1.elements();
                while ( rest.hasMoreElements() )
                {
                    String nextEvSeq = (String)rest.nextElement();
                    StringTokenizer evTok = new StringTokenizer( evSeq, "->" );
                    StringTokenizer nextEvTok = new StringTokenizer( nextEvSeq, "->" );
                    String firstEvAtt = evTok.nextToken();
                    String lastEvAtt = "";
                    while ( evTok.hasMoreTokens() )
                        lastEvAtt = (String)evTok.nextToken();
                    String firstNextEvAtt = (String)nextEvTok.nextToken();
                    if ( lastEvAtt.compareTo(firstNextEvAtt) == 0 )
                    {
                        //-------------------------------------------------------
                        //- handling transitivity, replace
                        //- A->B, B->C by A->C
                        //-------------------------------------------------------
                        String lastNextEvAtt = "";
                        while ( nextEvTok.hasMoreTokens() )
                            lastNextEvAtt = nextEvTok.nextToken();
                        String transitive = firstEvAtt + "->" + lastNextEvAtt;
                        temp2.removeElement(evSeq);
                        temp2.removeElement(nextEvSeq);
                        if (!temp2.contains(transitive) )
                            temp2.addElement(transitive);
                    }
                }
            }
            //-----------------------------------------------------------
            //- restoring 'evolve' hash table values
            //-----------------------------------------------------------
```

```
            Vector newEvol = (Vector)temp2.clone();
            Vector finalEvol = (Vector) temp2.clone();
            evolveTable.remove(obj);
            //-----------------------------------------------------------
            //- check template of new evolution sequences
            //- remove objects from evolveTable who have ALL
            //- sequences of the persisting template A->A
            //-----------------------------------------------------------
            Enumeration evSeqs = newEvol.elements();
            while ( evSeqs.hasMoreElements() )
            {
                String evSeq = (String)evSeqs.nextElement();
                if ( getTemplates(evSeq) == 0 )
                    finalEvol.removeElement(evSeq);
            }
            if ( finalEvol.size() != 0 )
                evolveTable.put(obj, finalEvol);
        }
    }

    public void prepareEvolveTable(Vector objs, String tplSeq)
    {
        //-----------------------------------------------------------
        //- preparing 'evolve' hash table
        //-----------------------------------------------------------
        for (int i=0; i<objs.size(); i++)
        {
            Vector evolVec = new Vector();
            String obj = (String)objs.elementAt(i);
            if ( !evolveTable.containsKey(obj) )
            {
                evolVec.addElement(tplSeq);
                evolveTable.put(obj, evolVec);
            }
            else
            {
                evolVec = (Vector)evolveTable.get(obj);
                evolVec.addElement(tplSeq);
            }
        }
    }

    public Vector getOtherProperties(Hashtable tplSeqTable)
    {
        Vector evolution = new Vector();
        Vector transition = new Vector();
        Vector result = new Vector();

        Enumeration tpl = tplSeqTable.keys();
        while ( tpl.hasMoreElements() )
        {
            String tplSeq = (String)tpl.nextElement();
            int pat = getTemplates(tplSeq);
            if ( pat == 1 )
            {
```

```
                Vector objs = (Vector)tplSeqTable.get(tplSeq);
                if ( objs.size() >= minSupport )
                    prepareEvolveTable(objs, tplSeq);
                else
                {
                    if ( (objs.size() < minSupport) && (objs.size() > 0) )
                    {
                        transition.addElement(objs);
                        transition.addElement(tplSeq);
                        transientProp.addElement(tplSeq);
                    }
                }
            }
        }

        //-----------------------------------------------------------
        //- organizing 'evolve' hash table values
        //- loading final values
        //-----------------------------------------------------------
        organizeEvolveTable(evolveTable);
        Enumeration evol = evolveTable.keys();
        while ( evol.hasMoreElements() )
        {
            String obj = (String)evol.nextElement();
            Vector evolVec = (Vector)evolveTable.get(obj);
            evolution.addElement(obj);
            evolution.addElement(evolVec);
            evPat.addElement(evolVec);
        }

        result.addElement(evolution);
        result.addElement(transition);
        return result;
    }
}
```

```
import java.util.*;

//-------------------------------------------------------------
//-  (C) Copyright 2001 Rabih A. Neouchi
//-------------------------------------------------------------
public class LCSObj extends TplObj
{
    String firstSeq;
    String fSeq;
    String secondSeq;
    String sSeq;
    String LCStr = "";
    String LCSub;
    int firstDim;
    int fDim;
    int secondDim;
    int sDim;
    int LCSeqLength[][];
    char optSol[][];

    LCSObj() {}
    LCSObj(String firstStr, String secondStr)
    {
        firstSeq = firstStr;
        fSeq = " " + firstSeq;
        firstDim = firstSeq.length();
        fDim = firstDim + 1;
        secondSeq = secondStr;
        sSeq = " " + secondSeq;
        secondDim = secondStr.length();
        sDim = secondDim + 1;
        LCSeqLength = LCSLength(fSeq, sSeq);
        printLCS(fSeq, firstDim, secondDim);
    }

    public int[][] LCSLength(String firstSeq, String secondSeq)
    {
        LCSeqLength = new int[fDim][sDim];
        optSol = new char[fDim][sDim];

        for (int i=1; i<fDim; i++)
            LCSeqLength[i][0] = 0;

        for (int j=0; j<sDim; j++)
            LCSeqLength[0][j] = 0;

        for (int i=1; i<fDim; i++)
        {
            for (int j=1; j<sDim; j++)
            {
                if ( fSeq.charAt(i) == sSeq.charAt(j) )
                {
                    LCSeqLength[i][j] = LCSeqLength[i-1][j-1]+1;
                    optSol[i][j] = 92;
                }
```

```
                else
                {
                    if ( LCSeqLength[i-1][j] >= LCSeqLength[i][j-1] )
                    {
                        LCSeqLength[i][j] = LCSeqLength[i-1][j];
                        optSol[i][j] = '^';
                    }
                    else
                    {
                        LCSeqLength[i][j] = LCSeqLength[i][j-1];
                        optSol[i][j] = '<';
                    }
                }
            }
        }
        return LCSeqLength;
    }

    public void printLCS(String fSeq, int i, int j)
    {
        if ( (i == 0) || (j == 0) ) return;
        if ( optSol[i][j] == 92 )
        {
            printLCS(fSeq, i-1, j-1);
            LCStr = LCStr + fSeq.charAt(i);
        }
        else
        {
            if ( optSol[i][j] == '^' )
                printLCS(fSeq, i-1, j);
            else
                printLCS(fSeq, i, j-1);
        }
    }
```

```
import java.util.*;

//-------------------------------------------------------------
//-  (C) Copyright 2001 Rabih A. Neouchi
//-------------------------------------------------------------
public class LMSObj extends TplObj
{
    String sequence;
    String sortedSeq;
    String LMStr;
    int seqLength;
    int seqAr[];
    int sortedAr[];

    LMSObj() {}
    LMSObj(String seqStr)
    {
        sequence = seqStr;
        seqLength = sequence.length();
        int seqAr[] = new int[seqLength];
        seqAr = convertSeq(sequence);
        int sotedAr[] = new int[seqLength];
        sortedAr = mergeSort(seqAr, seqLength);
        sortedSeq = convertSeq(sortedAr);
    }

    public int[] convertSeq(String seqStr)
    {
        int intAr[] = new int[seqStr.length()];

        for (int i=0; i<seqStr.length(); i++)
            intAr[i] = seqStr.charAt(i) - 48;

        return intAr;
    }

    public String convertSeq(int sortedAr[])
    {
        String result = "";

        for (int i=0; i<sortedAr.length; i++)
            result = result + sortedAr[i] + 48;

        return result;
    }

    public int[] mergeSort(int list[], int length)
    {
        int lower, upper, middle;
        int size;
        int ans[] = new int[list.length];

        // Set up the size of the trivial subarrays
        size = 1;
```

```
        // While we have not produced one sorted list of length length
        while (size < length) {

            // Merge as many pairs of sublists of length size as possible
            lower = 0;
            while (lower + size < length) {

            // Set up the bounds for this merge
                if (lower + size*2 >= length) {
                    upper = length-1;
                } else {
                    upper = lower + size*2 -1;
                }

                // Merge the subarrays
                ans = merge (list, lower, lower+size-1, upper);
                lower = upper + 1;
            }

            // Double the size of the subarrays to be merged for the next pass
            size *= 2;
        }
        return ans;
    }

    private int[] merge(int[] list, int first, int mid, int last)
    {
        int[] buffer = new int[last-first+1];
        int first1,first2, last1,last2, index;

        // Translate first, mid, last to two starts and finishes
        first1 = first; last1 = mid;
        first2 = mid + 1; last2 = last;

        // Set the next available location in the buffer array
        index = 0;

        // While both subarrays are non-empty copy the
        // smaller elements into the buffer.
        // Invariant: buffer[0..index-1] are sorted
        // smallest values from list[first..last]
        // and the remaining, larger values are in
        // list[first1..last1] and list[first2..last2] *)
        while (first1 <= last1 && first2 <= last2) {
            if (list[first1] < list[first2]) {
                buffer[index] = list[first1];
                first1++;
            } else {
                buffer[index] = list[first2];
                first2++;
            }
            index++;
        }
```

```
        // Now copy remainder of the non-empty one
        // Invariant: buffer[0..index-1] are the sorted smallest values from
        // list[first..last] and the remaining, larger values are in
        // list[first1..last1] or list[first2..last2]
        while (first1 <= last1) {
            buffer[index] = list[first1];
            first1++; index++;
        }
        while (first2 <= last2) {
            buffer[index] = list[first2];
            first2++; index++;
        }

        // Finally copy result back to parameter
        // Invariant: list'[first..index-1] are the sorted smallest values from
        // list[first..last]
        for (index = first; index <= last; index++) {
            list[index] = buffer[index-first];
        }

        return list;
    }

    public void mapLMSToLCS(String sequence, String sortedSeq)
    {
        LCSObj lcsObj = new LCSObj(sequence, sortedSeq);
        LMStr = lcsObj.LCStr;
    }
```

```
import java.util.*;
import java.io.*;

//---------------------------------------------------------
//-   (C) Copyright 2001 Rabih A. Neouchi
//---------------------------------------------------------
/**
 * This class can take a variable number of parameters on the command
 * line. Program execution begins with the main() method. The class
 * constructor is not invoked unless an object of type 'Form1'
 * created in the main() method.
 */
public class Form1 extends Form
{
    public Form1()
    {
        super();

        // Required for Visual J++ Form Designer support
        initForm();

        // TODO: Add any constructor code after initForm call
    }

    /**
     * Form1 overrides dispose so it can clean up the
     * component list.
     */
    public void dispose()
    {
        super.dispose();
        components.dispose();
    }

    private void bGo_click(Object source, Event e)
    {
        BufferedReader input = null;
        DataOutputStream output  = null;
        boolean endOfFileReached = false;
        String oneLine = "";

        //---------------------------------------------------------
        //- Use STEP algorithm to infer temporal properties
        //---------------------------------------------------------
        //---------------------------------------------------
        // Open patterns data files (*.DAT files)
        //---------------------------------------------------

        try  {input = new BufferedReader(new InputStreamReader
                                ((new FileInputStream( "patterns.idx" ))));}
        catch (Exception i)
        {
            System.out.println( "File not opened properly\n" + i.toString() );
            System.exit( 1 );
        }
```

```
try {output = new DataOutputStream(new FileOutputStream( "temprop.dat" ) );}
catch (Exception o)
{
    System.out.println( "File not opened properly\n" + o.toString() );
    System.exit( 1 );
}

while ( !endOfFileReached )
{
    try { oneLine = input.readLine(); }
    catch (Exception f) { try {input.close(); endOfFileReached = true;} catch (Exception f1) {} }
    StringTokenizer tokenizer = new StringTokenizer( oneLine, " " );

    String objName = tokenizer.nextToken();
    if ( objName.compareTo("end") == 0 ) endOfFileReached = true;
    int index = (new Integer(tokenizer.nextToken())).intValue();

    if ( index != 99 )
    {
        TplObj tplObj = new TplObj(objName, index);
        Vector objAllTimes = tplObj.processData(index);
        tplObj.preparePropTable(objAllTimes);
    }
    else
    {
        Patterns pat = new Patterns(1);
        //-----------------------------------------------
        //- Generate frequent 1-sequences
        //-----------------------------------------------
        Vector F1 = pat.generateF1(pat.propTable);
        if ( F1.isEmpty() ) break;
        else
        {
            System.out.println("//-- " + "F1 Sequences:");
            System.out.println(F1.toString());
            //-------------------------------------------
            //- Generate frequent 2-sequences
            //-------------------------------------------
            Vector F2 = pat.generateF2(F1);
            if ( !empty(F2) )
            {
                //System.out.println(pat.DB.toString());
                System.out.println("//-- " + "F2 Sequences:");
                System.out.println(F2.toString());
                System.out.println("//-- " + "F2 eqlSeqTable:");
                System.out.println(pat.eqlSeqTable.toString());
                System.out.println("//-- " + "F2 freqEqlSeqTable:");
                System.out.println(pat.freqEqlSeqTable.toString());
                System.out.println("//-- " + "F2 tplSeqTable:");
                System.out.println(pat.tplSeqTable.toString());
                System.out.println("//-- " + "F2 freqTplSeqTable:");
                System.out.println(pat.freqTplSeqTable.toString());
                //-----------------------------------------
                //- Generate frequent 3-sequences
                //-----------------------------------------
```

```
                Vector F3 = pat.generateF3(F2, F1);
                if ( !empty(F3) )
                {
                    System.out.println("//-- " + "F3 Sequences:");
                    System.out.println(F3.toString());
                    System.out.println("//-- " + "F3 eqlSeqTable:");
                    System.out.println(pat.eqlSeqTable.toString());
                    System.out.println("//-- " + "F3 freqEqlSeqTable:");
                    System.out.println(pat.freqEqlSeqTable.toString());
                    System.out.println("//-- " + "F3 tplSeqTable:");
                    System.out.println(pat.tplSeqTable.toString());
                    System.out.println("//-- " + "F3 freqTplSeqTable:");
                    System.out.println(pat.freqTplSeqTable.toString());
                    //-------------------------------------------------
                    //- Generate frequent 4-sequences
                    //-------------------------------------------------
                    Vector F4 = pat.generateF4(F3, F1);
                    if ( !empty(F4) )
                    {
                        System.out.println("//-- " + "F4 Sequences:");
                        System.out.println(F4.toString());
                        System.out.println("//-- " + "F4 eqlSeqTable:");
                        System.out.println(pat.eqlSeqTable.toString());
                        System.out.println("//-- " + "F4 freqEqlSeqTable:");
                        System.out.println(pat.freqEqlSeqTable.toString());
                        System.out.println("//-- " + "F4 tplSeqTable:");
                        System.out.println(pat.tplSeqTable.toString());
                        System.out.println("//-- " + "F4 freqTplSeqTable:");
                        System.out.println(pat.freqTplSeqTable.toString());
                    }
                }
            }
        }
        //-------------------------------------------------
        //- Determine persistent proprerties
        //-------------------------------------------------
        Vector persistProp = pat.getPersistentProperties(pat.freqTplSeqTable);
        //-------------------------------------------------
        //- Determine evolution patterns and transient properties
        //-------------------------------------------------
        Vector otherProp = pat.getOtherProperties(pat.tplSeqTable);
        Vector evPat = (Vector)otherProp.elementAt(0);
        Vector transientProp = (Vector)otherProp.elementAt(1);
        //-------------------------------------------------
        //- Output evolution patterns, persistent and transient properties
        //-------------------------------------------------
        try
        {
            output.writeBytes("Persistent Properties: " + "\n");
            output.writeBytes(persistProp.toString() + "\n");
            output.writeBytes("Evolution Patterns: " + "\n");
            output.writeBytes(evPat.toString() + "\n");
            output.writeBytes("Transient Properties: " + "\n");
            output.writeBytes(transientProp.toString() + "\n");
        }
```

```
                catch ( IOException io )
                {
                    System.err.println("Error during write to file\n" + io.toString());
                    System.exit( 1 );
                }
                //-----------------------------------------------------------
                //- Use TLAT algorithm to draw temporal edges
                //-----------------------------------------------------------
                TLAT tlat = new TLAT(persistProp, evPat);
                tlat.prepareEdges();
                tlat.drawEdges(tlat.tplEdgeTable);
                System.out.println(tlat.tplEdgeTable.toString());
            }
        }
    }

    /**
     * NOTE: The following code is required by the Visual J++ form
     * designer.  It can be modified using the form editor.  Do not
     * modify it using the code editor.
     */
    Container components = new Container();
    Button bGo = new Button();

    private void initForm()
    {
        this.setText("Inferring Temporal Properties");
        this.setAutoScaleBaseSize(new Point(5, 13));
        this.setClientSize(new Point(300, 300));

        bGo.setLocation(new Point(112, 120));
        bGo.setSize(new Point(75, 23));
        bGo.setTabIndex(0);
        bGo.setText("Extract");
        bGo.addOnClick(new EventHandler(this.bGo_click));

        this.setNewControls(new Control[] {
                                bGo});
    }

    /**
     * The main entry point for the application.
     *
     * @param args Array of parameters passed to the application
     * via the command line.
     */
    public boolean empty(Vector F2)
    {
        Vector eql = (Vector)F2.elementAt(0);
        Vector tpl = (Vector)F2.elementAt(1);
        if ( eql.isEmpty() && tpl.isEmpty() )
            return true;
        return false;
    }
```

```
    public static void main(String args[])
    {
        Application.run(new Form1());
    }
}
```

Appendix B
TLAT Algorithm Code

```
import java.util.*;

//-------------------------------------------------------------
//-  (C) Copyright 2001 Rabih A. Neouchi
//-------------------------------------------------------------

//-------------------------------------------------------------
//-  TLAT - Temporal LATtices
//-------------------------------------------------------------
public class TLAT extends TplObj
{
    Hashtable tplEdgeTable = new Hashtable();
    Vector persistProp = new Vector();
    Vector evPat = new Vector();

    TLAT() {}
    TLAT(Vector persistProp, Vector evPat)
    {
        this.persistProp = (Vector)persistProp.clone();
        this.evPat = (Vector)evPat.clone();
    }

    public void prepareEdges()
    {
        //-------------------------------------------------
        //- Preparing directed circular edges
        //-------------------------------------------------
        for (int i=0; i<persistProp.size(); i+=2)
        {
            String att = (String)persistProp.elementAt(i);
            if ( !tplEdgeTable.containsKey(att) )
                tplEdgeTable.put(att, (Vector)persistProp.elementAt(i+1));
        }

        //-------------------------------------------------
        //- Preparing directed edges from one node to another
        //-------------------------------------------------
        for (int i=0; i<evPat.size(); i+=2)
        {
            String obj = (String)evPat.elementAt(i);
            Vector evolSeqs = (Vector)evPat.elementAt(i+1);

            String objSeq = "";
            for (int j=0; j<evolSeqs.size(); j++)
            {
                Vector evolVec = new Vector();
                String evolSeq = (String)evolSeqs.elementAt(j);
                StringTokenizer tok = new StringTokenizer(evolSeq, "->");
```

```
                while ( tok.hasMoreTokens() )
                {
                    String att = tok.nextToken();
                    Vector objIDs = (Vector)propTable.get(att);
                    String ID = (String)objIDs.elementAt(objIDs.indexOf(obj)+1);
                    String aliasName = obj + ID;
                    objSeq = objSeq + "->" + aliasName;
                }
                objSeq = objSeq.substring(2, objSeq.length());
                if ( !tplEdgeTable.containsKey(evolSeq) )
                {
                    evolVec.addElement(objSeq);
                    tplEdgeTable.put(evolSeq, evolVec);
                }
                else
                {
                    evolVec = (Vector)tplEdgeTable.get(evolSeq);
                    evolVec.addElement(objSeq);
                }
            }
        }
    }

    public void drawEdges(Hashtable tplEdgeTable)
    {
        Enumeration edges = tplEdgeTable.keys();
        while ( edges.hasMoreElements() )
        {
            String edge = (String)edges.nextElement();
            int idx = edge.indexOf("->");
            if ( idx == -1)
            {
                //-------------------------------------------------
                //- Draw circular directed edges
                //-------------------------------------------------
                Look in the concept lattice for the node whose attribute label = edge;
                Draw a circular directed edge around that node;
            }
            else
            {
                //-------------------------------------------------
                //- Draw regular directed edges
                //-------------------------------------------------
                Vector edgeObjs = (Vector)tplEdgeTable.get(edge);
                if ( edgeObjs.size() > 1)
                {
                    //-------------------------------------------------
                    //- The edge is assigned to more than one evolving object
                    //- Draw the edge for these objects based on attribute labels
                    //- Draw the temporal edge as high as possible
                    //-------------------------------------------------
                    StringTokenizer attTok = new StringTokenizer(edge, "->");
                    String previousAttLabel = "null";
```

```
                    while ( attTok.hasMoreTokens() )
                    {
                        String attLabel = attTok.nextToken();
                        Look in the concept lattice for the node whose attribute label = attLabel;
                        Draw a regular directed edge from the concept node
                        whose attribute label = previousAttLabel to the current node
                        whose attribute label = attLabel;
                        previousAttLabel = attLabel;
                    }
                }
                else
                {
                    //-----------------------------------------------------
                    //- The edge is assigned to only one evolving object
                    //- Draw the edge for that object based on object labels
                    //-----------------------------------------------------
                    Enumeration edgeObjSeqs = edgeObjs.elements();
                    String objLabels = (String)edgeObjSeqs.nextElement();
                    while ( edgeObjSeqs.hasMoreElements() )
                    {
                        StringTokenizer objTok = new StringTokenizer(objLabels, "->");
                        String previousObjLabel = "null";
                        while ( objTok.hasMoreTokens() )
                        {
                            String objLabel = objTok.nextToken();
                            Look in the concept lattice for the node whose object label = objLabel;
                            Draw a regular directed egde from the concept node
                            whose object label = previousObjLabel to the current node
                            whose object label = objLabel;
                            previousObjLabel = objLabel;
                        }
                    }
                }
            }
        }
    }
}
```

Vita Auctoris

Rabih Ahmad Tarek Neouchi was born in Tripoli, a city in North Lebanon, Lebanon on June 8, 1975. He graduated from Rawdat Al-Fayhaa High School in Tripoli, Lebanon in June 1993 with a Lebanese Baccalaureate II degree (Experimental Sciences). From there he went to the Lebanese American University (LAU), Beirut, Lebanon where he obtained a B.Sc. degree in Computer Science in June 1997. He immigrated to Canada in 1998. In September 1999, Rabih joined the Master of Science Program in Computer Science at the University of Windsor, Windsor, Ontario, Canada. Rabih is planning to pursue a Doctor of Philosophy degree in Computer Science at the Computing Science Department at the University of Alberta, Edmonton, Alberta, Canada where he has been admitted, and will be joining the newly founded RoboCup team (Team CANUCK[5]). His current research interests lie in the area of Artificial Intelligence (AI), and more specifically robotics, planning, agent communication policies, human cognition, temporal reasoning, and lattice theory.

[5] http://www.cs.ualberta.ca/~robotics/robocup/